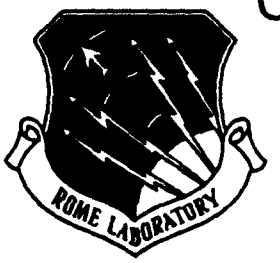
(1)

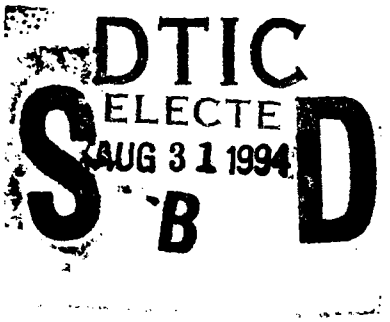
RL-TR-94-57
Final Technical Report
May 1994

AD-A283 914



EVALUATION AND DEVELOPMENT OF MULTIMEDIA NETWORKS IN DYNAMIC STRESS (EDMUNDS)

SRI International



Sponsored by
Advanced Research Projects Agency
ARPA Order No. 6967

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

Rome Laboratory
Air Force Materiel Command
Griffiss Air Force Base, New York

This report has been reviewed by the Rome Laboratory Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

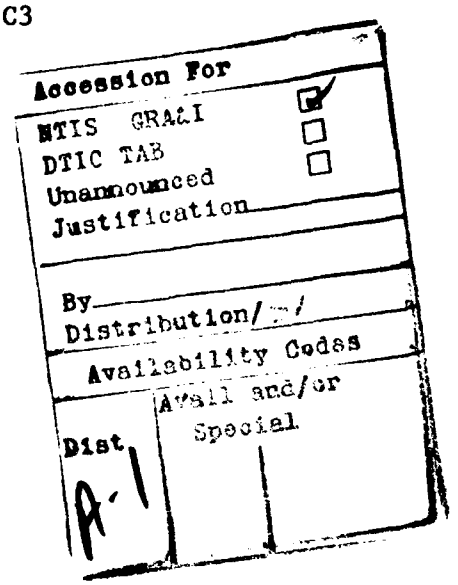
RL-TR-94-57 has been reviewed and is approved for publication.

APPROVED:

CHARLES MEYER
Project Engineer

FOR THE COMMANDER

JOHN A. GRANIERO
Chief Scientist for C3



If your address has changed or if you wish to be removed from the Rome Laboratory mailing list, or if the addressee is no longer employed by your organization, please notify RL (C3BC) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

EVALUATION AND DEVELOPMENT OF MULTIMEDIA NETWORKS IN DYNAMIC STRESS (EDMUNDS)

John Hight
Edmond Costa
Diane Lee
Richard Ogier
Julie Wong

Contractor: SRI International
Contract Number: F30602-90-C-0003
Effective Date of Contract: 26 Dec 89
Contract Expiration Date: 26 Jul 93
Short Title of Work: EDMUNDS
Period of Work Covered: Dec 89 - Jul 93

Principal Investigator: John Hight
Phone: (415) 859-4279

RL Project Engineer: Charles Meyer
Phone: (315) 330-1880

This research was supported by the Advanced Research Projects Agency of the Department of Defense and was monitored by Charles Meyer, RL (C3BC), 525 Brooks Road, Griffiss AFB NY 13441-4505, under Contract F30602-90-C-0003.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	May 1994	Final Dec 89 - Jul 93

4. TITLE AND SUBTITLE
EVALUATION AND DEVELOPMENT OF MULTIMEDIA NETWORKS IN DYNAMIC STRESS (EDMUNDS)

5. FUNDING NUMBERS
C - F30602-90-C-0003
PE - 62708E
PR - F967
TA - 00
WU - 01

6. AUTHOR(S)
John Hight, Edmond Costa, Diane Lee, Richard Ogier, Julie Wong

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
SRI International
333 Ravenswood Avenue
Menlo Park CA 94025-3493

PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Advanced Research Projects Agency
3701 North Fairfax Drive
Arlington VA 22203-1714

Rome Laboratory (C3BC)
525 Brooks Road
Griffiss AFB NY 13441-4505

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
RL-TR-94-57

11. SUPPLEMENTARY NOTES
Rome Laboratory Project Engineer: Charles Meyer/C3BC/(315) 330-1880

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

Existing network protocols cannot satisfy the requirements of future command, control, and communications (C3I) applications over dynamic, stressed networks with multiple transmission media (multimedia). EDMUNDS is a project directed by Rome Laboratory (RL) under the Advanced Research Projects Agency's (ARPA's) Secure Tactical Internet (STI) Program, which is aimed at advancing the state of survivable network protocols in order to provide the foundation required for these future tactical networks. First, descriptions of the EDMUNDS development environments are presented in Sections 2 through 5; we then describe the STI protocols developed under EDMUNDS, in Section 6; the benchmark protocols used in analyzing the STI protocols are described in Sections 7 and 8; and our emulator development efforts are described in Section 9. Finally, we present concluding remarks in Section 10.

The appendices to this report are as follows: "The Network Environment Profile," (Appendix A); "How to Execute a NAPI + Simulation," (Appendix B); "How to Analyze a NAPI + Simulation," (Appendix C); "How to Implement a Protocol in NAPI +," (Appendix D); a report on the Secure Tactical Internet Protocol (STIP)3 (Appendix E); and finally, three papers published under the EDMUNDS project (Appendix F).

14. SUBJECT TERMS
Multimedia, Dynamic Stress, Algorithms, Protocols

15. NUMBER OF PAGES
258

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

CONTENTS

FIGURES

EXECUTIVE SUMMARY

OBJECTIVES

In response to rapid changes in the tactical environment, the military services are creating a new generation of tactical communication networks and processing environments. These changes are driving the requirements for future C^3I environments. The characteristics and operating conditions of these environments that affect new protocol development for these tactical networks include

- **Frequent Topological Changes.** These changes can be planned (e.g., node movement and topology control); random (e.g., environmental effects); or due to hostile forces (e.g., node destruction and electronic warfare systems for manning, wiretapping, direction finding, and traffic analysis).
- **Diverse Communication Media.** To increase system survivability, the network elements must be interconnected by a multitude of communication links and networks, including cable, fiber optics, and radio channels of diverse bandwidth characteristics, directionality, and coding schemes. These links will vary widely with respect to throughput, delay, and reliability.
- **Diverse Communication Requirements.** The network will need to support many different types of messages, including speech, video, sensor data, messages for distributed data processing, and firing commands. These different types differ widely in their delay, throughput, and reliability requirements, so the network must provide *multiple types of services* (TOS's).
- **Sudden Changes in Communication Load.** For example, the load would increase dramatically at the onset of a battle.
- **Networks Size.** The number of nodes and end users and the distance over which they are dispersed will increase. Access must be provided to a large number of sensors, weapons, and command and control centers dispersed over large areas. This access must allow the rapid deployment of forces, mobility, and interoperability.

Current technology can only support networks with some, but not all of the above characteristics. Significant technological advances must be made before the new generation of networking is realizable. Thus, the Evaluation and Development of Multimedia Networks in Dynamic Stress (EDMUNDS) project, which is directed by Rome Laboratory (RL) under the Advanced Research Project Agency's (ARPA's) Secure Tactical Internet (STI) program, is aimed at advancing the state of survivable network protocols in order to provide the foundation required for these future tactical networks. The purpose of EDMUNDS research was to evaluate and develop algorithms for multimedia networks operating under highly dynamic conditions. SRI International (SRI) investigated the causes (random, malicious, and controlled) of network dynamics, quantitatively evaluated transient network behavior, and developed algorithms that permit effective service for C^3I applications over multimedia networks, despite these conditions. To accomplish these tasks, we used a comprehensive, integrated approach to the development and evaluation of dynamic multimedia network algorithms and protocols.

APPROACH

SRI's approach for achieving the goals of the EDMUNDS project was to divide the program into the following primary tasks:

- **Task 1: Develop Robust Network Routing Algorithms.** Develop theoretically sound, fast-responding, efficient, and robust network algorithms for multimedia networks. These new algorithms were being developed to provide contingency plans to react immediately in case of sudden changes, to effectively utilize the resources of multiple transmission media, to make use of redundancy for improved reliability and robustness in hostile environments, and to combine routing and flow control to satisfy the diverse requirements of C^3I traffic.
- **Task 2: Develop a Modular Protocol Framework.** Design a modular protocol framework to facilitate the comparison of protocols composed of different sets of algorithms. Upon this framework, develop a series of increasingly sophisticated protocols to evaluate and demonstrate the algorithms developed in Task 1.
- **Task 3: Develop an Integrated Network Evaluation Environment.** Develop an integrated set of network simulation and emulation tools to support the phased development and evaluation of the protocols. Develop an emulator over the Mach operating system to take advantage of parallel computation facilities and to ease the porting of the emulator and the emulated protocols to other systems supporting Mach.
- **Task 4: Develop Red-Team Strategies and Performance Measures.** Develop methods to quantify the performance of the protocols under both benign and stressful ECM conditions. Develop and evaluate ECM strategies to intelligently attack network operation by using knowledge of the network protocols. Feed back the results into the development cycle.

ACCOMPLISHMENTS

The following subsections detail the main accomplishments of this project.

Secure Tactical Internet Protocols

Under the EDMUNDS project, SRI developed the Secure Tactical Internet Protocols (STIP), a series of three increasingly efficient and robust protocols. STIP1 was the first of the series, representing an exciting new direction for survivable routing/flow control protocols for the tactical environment. Specifically, STIP1 was developed, using a theoretical analysis, to provide high-throughput, low-delay, fair service in multimedia networks that are subject to jamming. The protocol makes use of alternate routing to select outgoing links (among all media), based on local information about dynamic link states and distributed information about congestion.

STIP2 improved upon STIP1 by using several new methods for better performance and increased robustness against a "protocol-smart" adversary. These improvements include a new algorithm for maintaining the Directed Acyclic Graph (DAG); an improved flow-computation algorithm that predicts and avoids congestion; a method for spreading traffic over multiple paths for increased robustness to jamming; and an improved link scheduling algorithm.

STIP3 improved upon STIP2 by using a faster-responding link probability estimator, an improved link scheduler that supports multiple priorities, and a flow computation algorithm that optimizes a global rather than a local objective.

The STIP protocols are based on a distance-vector algorithm*; that is, they belong to the class of distributed routing algorithms in which each node *i* informs its neighbors of the delay from itself to every destination. The distributed Bellman-Ford shortest-path algorithm is another example of a distance-vector algorithm.† However, the Bellman-Ford algorithm is a single-path algorithm, while all the current STIP protocols are based on multipath algorithms.

In order to gauge the performance and behavior of the STIP protocols, SRI constructed a simple baseline protocol (Baseline). Baseline shares the same link computations and maintenance, expected delay smoothing and thresholding, and queue prioritization algorithms that were introduced during the project for STIP1. However, Baseline is based (in part) on Bellman-Ford, and uses simple forwarding and alternate routing rules in place of the sophisticated flow computation and link scheduling algorithms used in STIP1.

Simulations have shown that *STIP3 performs better than the Baseline protocol in most experiments, and in many cases is far superior*. Baseline was found to have as much as 6.7 times the mean end-to-end delay and as much as 3.7 times the response time of STIP3.

STIP3 is appropriate for use in a variety of networks, including applications using point-to-point and microwave links; in radio networks (with modifications); and in the Internet.

Future challenges for STI protocol research include

- The use of global topology information, especially for faster links, to
  - Route on disjoint paths
  - Avoid being limited to a single DAG
  - Avoid loops
- ATM compatibility
- Bandwidth reservations and admission control
- Alternate path reservations for reliable service
- Erasure coding for recovery of lost packets
- Dynamic adjustment of protocol parameters
- The incorporation of transmission scheduling for radio networks
- The use of neural networks for faster computation.

NAPI+ Environment

The Network Algorithm Programmer's Interface Plus (NAPI+) is an object-oriented specification for developing network protocols. The NAPI+ environment can be used to implement new protocols efficiently (without having to develop all the baggage of a full protocol

*Schwartz, M. 1986. *Telecommunications Networks: Protocols, Modeling and Analysis*, Addison-Wesley Publishing Co., Reading, Massachusetts.

†Schwartz 1986; Bertsekas, D., and R. Gallager. 1987. *Data Networks*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey.

implementation). Implementing a protocol so that it is compliant with NAPI+ permits a rapid progression from simulation to full packet-switch implementation by allowing the use of a single protocol module for both efforts

We have written a NAPI+ specification that details the interface between a protocol-independent environment module and a protocol module. In addition to designing the NAPI+ specification and implementing a NAPI+ environment module, we implemented four protocol modules and integrated them with the environment module: (1) Baseline (an early benchmark protocol); (2) STIP2; (3) STIP3; and (4) OSPF.

Network Environment Profile

In order to guide the protocol development and analysis efforts, SRI identified a framework of network conditions of concern to EDMUNDS. This framework, along with a methodology for determining the performance of protocols, was presented in the reports *EDMUNDS Network Environment Profile: Benign and Adversarial Conditions, and Benchmark Scenarios*, Parts 1 and 2.*

The Network Environment Profile focuses on link speeds of up to 10 Mbps. The profile considers networks of moderate size (50 to 1000 nodes); link dynamics where the jamming period is less than a packet size; and various types of traffic, such as voice, file transfer, database updates, data messages, and sensor traffic. Future challenges would include extending the profile to a more general model and applying it to higher-speed and larger multiple-media networks.

Red-Teaming

SRI created a number of tools to enable the experimenter to simulate the effects of a malicious adversary who is jamming nodes and links. Jamming is simulated by the specification of link dynamics, i.e., scheduling the up and down states of a link. By changing the link dynamics, the experimenter can measure the effect of an adversary's jamming attack.

Analysis Tools

Tools were created to automate the execution of several simulation experiments in sequence. The ability to batch several experiments together is especially useful for unattended overnight operation and for exploring parameter sensitivity.

Several tools were also developed specifically to help with protocol behavior analysis. These include software tools to measure and plot the end-to-end delay of user-traffic, software to perform Measures of Effectiveness (MOEs), and software to graphically animate network packet transmissions. The design of these tools was guided by the Network Environment Profile.

The analysis tools developed under EDMUNDS are applicable to any network with point-to-point links. These tools could also be extended to handle networks of other types (e.g., broadcast and multiple access).

---

*Lee, D.S., D.A. Beyer, and R.G. Ogier. 1992(a). *EDMUNDS Network Environment Profile: Benign and Adversarial Conditions, and Benchmark Scenarios (Part I)*, ITAD-8558-TR-91-23, SRI International, Menlo Park, California (May). Lee. D.S., D.A. Beyer, and R.G. Ogier. 1992(b). *EDMUNDS Network Environment Profile: Benign and Adversarial Conditions, and Benchmark Scenarios (Part 2)* ITAD-8558-TR-91-23, SRI International, Menlo Park, California (July), SECRET.

Traffic Generation Module

The DARTNET Traffic Generator (TG) software provides a convenient method for specifying the characteristics of user traffic. Traffic streams are specified by the use of parameterized distribution models (e.g., constant, uniformly random, exponential) for scheduling packet generation. These distributions are used to control interpacket gaps, interburst gaps, and packet lengths.

The DARTNET TG software was modified and integrated with the NAPI+ environment and is a part of all NAPI+ simulations. The Network Environment Profile was used to guide this integration.

The distribution models in the TG software were developed on the assumption of link speeds of under 10 Mbps. The Traffic Generator could be extended to include models that are more appropriate for high-speed networks.

Papers Published

Three papers were published under EDMUNDS for INFOCOM:

- Ogier, R.G., and V. Rutenbur_ 1992(a). "Minimum Expected Delay Alternate Routing (MEDAR)," INFOCOM '92.
- Ogier, R.G., and V. Rutenburg. 1992(b). "Robust Routing for Minimum Worst Case Delay in Unreliable Networks, INFOCOM '92.
- Rutenburg, V., and R.G. Ogier. 1993. "How to Extract Maximum Information from Event-Driven Topology Updates." INFOCOM '93.

The first two papers address the issue of providing precomputed optimal alternate paths in case of node or link failures in a network. The third paper presents an efficient method for disseminating link state information. All three papers are provided in Appendix F of this report.

1 INTRODUCTION

Existing network protocols cannot satisfy the requirements of future command, control, and communications (C^3I) applications over dynamic, stressed networks with multiple transmission media (multimedia). EDMUNDS* is a project directed by Rome Laboratory (RL) under the Advanced Research Project Agency's (ARPA's) Secure Tactical Internet (STI) Program, which is aimed at advancing the state of survivable network protocols in order to provide the foundation required for these future tactical networks.

This document is the final technical report for the EDMUNDS Project by SRI International (SRI) under Contract F30602-90-C-003 for ARPA and Rome Laboratory.

First, descriptions of the EDMUNDS development environments are presented in Sections 2 through 5; we then describe the STI protocols developed under EDMUNDS, in Section 6; the benchmark protocols used in analyzing the STI protocols are described in Sections 7 and 8; and our emulator development efforts are described in Section 9. Finally, we present concluding remarks in Section 10.

The appendices to this report are as follows: "The Network Environment Profile," (Appendix A); "How to Execute a NAPI+ Simulation," (Appendix B); "How to Analyze a NAPI+ Simulation," (Appendix C); "How to Implement a Protocol in NAPI+," (Appendix D); a report on the Secure Tactical Internet Protocol (STIP) 3 (Appendix E); and finally, three papers published under the EDMUNDS project (Appendix F).

*EDMUNDS: Evaluation and Development of Multimedia Networks in Dynamic Stress.

2 SIMPLE NETWORK ALGORITHM PROTOTYPING SIMULATOR

The STIP1 OPNET implementation provided discrete event simulation to any desired resolution, accurate channel modelling, sophisticated control and data probing, and built-in graphing capabilities for testing and evaluating network protocols. However, researchers required a significant start-up effort, to become familiar with using such a simulation:

- Learning a new graphic user interface
- Reading through and learning the detailed user's manuals
- Installing environment files
- Transferring the OPNET license to a local machine or working on a centrally shared machine (OPNET did not have a networked license server at the time)
- Implementing not only core algorithms, but a significant number of other protocol components (e.g., acks, control packets) so that the algorithms could be tested.

For someone who spends a significant proportion of his or her time running and implementing protocol simulations, the advantages of the commercial simulation package OPNET are likely to far outweigh the start-up effort. This is likely not the case for someone whose primary responsibilities are to theoretically analyze and design algorithms rather than implement whole protocols. Still, the initial prototyping and testing of core network algorithms is best performed by the algorithm developers. For these people, we developed the Simple Network Algorithm Prototyping Simulator (SNAPS) with the following characteristics in mind. Specifically, the simulator was designed to

- Require minimal time to learn and use
- Permit rapid prototyping and testing of core network algorithms (without the potential bugs, interrelationships, sensitivities, etc., of an entire protocol package)
- Permit credible comparisons between competing algorithms in different network scenarios
- Require no license or environment files other than those for programming in C
- Require only a few pages of documentation.

The following features are provided by SNAPS:

- SNAPS is based on C.
- The interface to SNAPS consists of a total of only about a dozen user and SNAPS functions.
- SNAPS permits flexible specification of the following link characteristics:
  - Capacity
  - Propagation delay
  - Retransmission timeout
  - Link dynamics variables
- SNAPS produces logging output that is compatible with the analysis tools developed during the EDMUNDS project.

- SNAPS provides an interface to the SURAN Standard Scenario (SSS) Generator. The SSS Generator was developed during the SURAN project to facilitate the creation of nonpartitioned topologies of specific density, mobility, and jamming intensity.*

2.1 MODEL

SNAPS was developed according to the following model:

- Time is slotted. The maximum link capacity is one packet per slot.
- Probe packets are not simulated. Nodes can estimate the link probabilities by monitoring and smoothing the actual link states.
- Ack packets are not simulated (and thus consume no link capacity). Packet success is randomly computed (based on link state) at the time of packet transmission. Unsuccessful packets are enqueued on a timeout queue, to be retransmitted after the retransmission timeout delay for that link.
- Update packets are not simulated (and thus consume no link capacity). Nodes "learn" of information to be reported from a neighbor, K slots after the neighbor has updated information.
- All packets have the same length.

Since it provides less detailed simulations than the STIP1 OPNET implementation, SNAPS can be used for a quick evaluation of the main principles of the algorithms tested, separate from complicating details.

2.2 MAIN PROGRAM

The user is responsible for writing the main program loop and declaring the variables pertinent to the algorithm, including the packet queues. The main program should resemble the following:

```
int main()
{
    Init_Tools(argc, argv);      /* SNAPS function */
    Init_All_Links();            /* Scenario file function */
    Init_Packet_Queues();        /* User function to create all  */
                                 /* packet queues needed by the  */
                                 /* algorithm by calling         */
                                 /* New_Queue() for each one.    */
    Init_Algorithm_Variables(); /* User function. */
```

*Beyer, D.A. *SURAN Standard Scenario (SSS) Generator User's Guide* [SURAN Program Technical Note 66], SRI International, February, 1991.

```
    /* Run simulation for 1000 slots. */
    for (slot = 0; slot < 1000; slot++) {
        Update_Link_States();    /* comment Scenario file function */
        Algorithm_Processing(); /* comment User function */
        /* Simulate propagation of control  */
        /* information every 10 slots.      */
        if (slot%10) {
            Update_Processing();     /* User function */
        }
        Generate_Pkts(1.0);      /* Scenario file function */
        Process_Links();         /* SNAPS function */
    }
}
```

Additionally, the user/developer software must declare the global variable **int slot** and use it to indicate the simulation time.

2.3 INTERFACE TO SNAPS

This section describes the entire interface to SNAPS. The interface is divided into two sections, one for SNAPS functions that are called by the user's code, and the other for user functions called by the SNAPS library.

2.3.1 SNAPS Functions

- **int Init_Tools(int argc, char * argv[])**
  Initializes the SNAPS library; this function must be called once at the beginning of the program, before any other SNAPS library function is called. argv[1] should be set to the name of the file for the SNAPS output logging; otherwise (if argc < 2) the user is prompted for the name of the log file. The other arguments (argv[2] through argv[50]) will be interpreted as floats and will be used to initialize the global **int num_args** and the global array **float arg[50]**. The function returns SUCCESS (1) or FAILURE (0).
- **int Init_Link(int from, int to, int inv_capacity, int prop_delay, int timeout, int p1, double p2, doublep3)**
  Initializes a directed link between the specified nodes (from and to) with the specified inverse capacity (in slots/packet), propagation delay (in slots), and retransmission timeout (in slots). The variables **p1**, **p2**, and **p3** are used by the user to keep track of the current state of the link and the link's dynamics. The function returns SUCCESS or FAILURE.
- **int Init_SSS_Scenario(int num_nodes, int density, int seed)**
  Makes the appropriate calls to **Init_Link()** to set up a nonpartitioned scenario of specified density (low, medium, or high), using the random number seed specified.
- **int Process_Links()**
  The main SNAPS function to perform all processing that involves packet transmissions, propagations, timeouts, and link-available notifications.

- **PKT * Nth_Pkt_On_Q(PKT_QUEUE *q, int *n)**
  Retrieves (without dequeueing) a pointer to the nth packet on the queue passed, or the last packet if there are fewer than n on the queue. The value of n is set to the actual position of the referenced packet.
- **int Generate_Pkt(int source, int destination)**
  Generates a packet at the source and to the destination specified; returns SUCCESS or FAILURE.
- **int Num_Links(int node)**
  Returns the number of links for the given node.
- **LINK * Get_Link(int node, int link_num)**
  Returns a pointer to the link structure for the given node and link number.
- **PKT_QUEUE * New_Queue()**
  Returns a new packet queue or NULL upon failure.

2.3.2 User Functions

- **int Log_Received_Pkt(PKT *p)**
  Called when a packet is received at its final destination to permit the user to update end-to-end traffic statistics; should return SUCCESS or FAILURE.
- **PKT_QUEUE * Packet_Arrival(PKT *p, int type)**
  Called when a packet arrives at a node that is not its final destination. The type argument, {GENERATING, FORWARDING, TIMEOUT}, indicates that the packet has just been generated at this node, has just been received, and should be forwarded by this node; or should just be timed out from a past transmission by this node. The user should return a pointer to the packet queue on which to enqueue the packet.
- **PKT * Link_Available(LINK *link)**
  Called by the **Process_Links()** function for all links that are available to transmitting a packet. The user should return a pointer to the packet to be transmitted or NULL. (The pointer to the packet is typically retrieved using **Nth_Pkt_On_Q()**.)
- **int Pkt_Success(LINK *link)**
  Called at the start of transmitting a packet on a link to find out whether the packet transmission will be successful. The user should return either SUCCESS or FAILURE.

2.4 MACROS AND PACKET, LINK, AND QUEUE DATA TYPES

This section lists the key macros and data structure types defined by the SNAPS library.

```
#define NULL                    0
#define FAILURE                 0
#define SUCCESS                 1
#define MAX_NUM_NODES           100
#define MAX_NUM_LINKS_PER_NODE  10
/* Types of arriving packets */
#define GENERATING              0
#define FORWARDING              1
#define TIMEOUT                 2

typedef struct {
    int     src;        /* source node */
    int     dst;        /* destination node */
    int     cur_node;   /* Nnode that this packet is at currently */
                        /* Equal to -1 if pkt is propagating */
    int     seq;        /* Seq number */
    int     gtime;      /* Generation time */
    int     atime;      /* Arrival time (time it was originally */
                        /* enqueued at current node) */
    int     next_time;  /* pkt is eligible for retransmission after */
                        /* (next_time < slot) */
    int     rexmt_cnt;  /* used in baseline */
    PKT_QUEUE *cur_queue; /* The q the pkt is sitting on */
} PKT;

typedef struct {
    int     from;       /* tail node of link */
    int     to;         /* head node of link                        */
    int     link_num;   /* link_num of from node                    */
    int     capacity;   /* in slots to xmt a pkt (actually inv. cap.*/
    int     prop_delay; /* in slots to propagate a pkt              */
    int     timeout;    /* slots that a pkt must wait before rexmt  */
    int     p1;         /* link dynamics fields                     */
    double  p2;         /* e.g., link state, link probability,      */
    double  p3;         /*    or Markov variables                   */
    PKT     *in_transit; /* This is the packet being transmitted    */
    int     xmt_time;   /* >0 if a packet is in transit             */
                        /* can be interpreted as link_busy          */
    PKT_QUEUE *prop_pkts; /* Propagating packets. These packets */
                          /* have completed transmission but     */
                        /* have not yet been completely         */
                        /* received. They may be completely in  */
                        /* the ether.                           */
} LINK;

typedef struct {
    int         size;       /* number of packets */
    Q_ELEMENT   *head;      /* packet with earliest arrival time */
    Q_ELEMENT *tail;        /* Latest packet */
} PKT_QUEUE;
```

3 NETWORK ALGORITHM PROGRAMMER'S INTERFACE

As previously mentioned, the SNAPS tool provided a simulation environment that required minimal time to learn and permitted the rapid prototyping and testing of algorithms without the excess baggage of an entire protocol package (such as the STIP1-OPNET simulation). However, the testing of algorithms eventually progresses to a point where a fully detailed protocol simulation is warranted. At this point, the algorithm developer is still faced with the tasks of learning how to use the OPNET tool, porting the algorithm code to a format suitable for use with the OPNET software library, and copying all the protocol-independent parts of the STIP1-OPNET simulation to each new protocol package built. In order to meet the need for a mechanism for the transition to a full protocol implementation, we designed the Network Algorithm Programmer's Interface (NAPI).

3.1 TRANSITIONING FROM SNAPS TO NAPI

We wanted to provide a simulation environment with a simple function specification similar to that of the SNAPS tool, but independent of underlying simulation packages (OPNET or SNAPS). The goal was to allow the protocol developer to write only one protocol implementation, which could be linked with a SNAPS-like environment or an environment with the full detail of the STIP1-OPNET simulation.

We created such a function specification and named it NAPI. Using the protocol-independent sections of the SNAPS tool and the STIP1-OPNET simulation, we created two NAPI-compliant environment modules: the NAPI-SNAPS and NAPI-OPNET environments. Once the NAPI-SNAPS and NAPI-OPNET environment modules were built, the algorithm developer could write just one NAPI-compliant protocol module that could be used with both modules.

Prior to the creation of NAPI, the initial STIP2 protocol implementation was done in SNAPS, while the EDMUNDS Baseline protocol was implemented both in OPNET and in SNAPS. We ported both the STIP2 and Baseline implementations to the new NAPI format, and linked each module with both the NAPI-SNAPS and the NAPI-OPNET environments. Figure 1 shows the relationships of the STIP1 simulation, the SNAPS tool, and the four new NAPI simulations.

3.2 OVERVIEW OF NAPI COMPONENTS

NAPI consists of the following main components:

- **Environment**. Environment (Env) or simulation platform. The Environment is mostly protocol independent.
- **Core**. Network algorithm (protocol-dependent) part.
- **Wrapper**. The Wrapper (Wpr) is the part of the Environment that changes as the Core protocol changes. For example, functions for manipulating update packets are contained in the Wrapper.
- **Libraries**. Other libraries containing the Traffic Generator (TG) and Parameter File Parser (PFP).

Figure 2 illustrates the relationships between these main components.

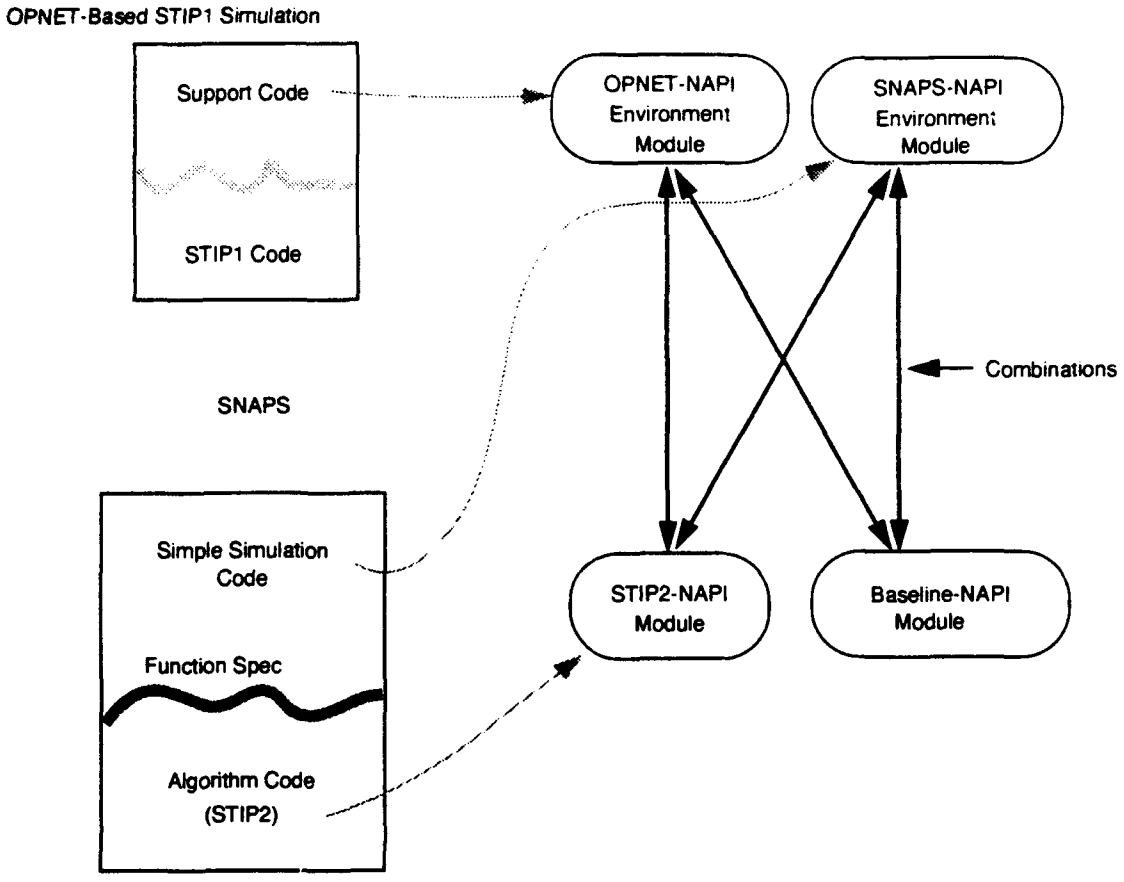


Figure 1. Porting from OPNET and SNAPS

3.3 ENVIRONMENT COMPONENT

The Environment is responsible for

- Queue management and default packet prioritization
- Computation of link timeout windows and link probabilities
- Ack and probe packet generation and processing
- Packet transmission, transmission list management, and retransmission processing
- Logging of simulation data
- Keeping the simulation clock
- Interfacing with the TG library and PFP library.

Environment functions are described below.

3.3.1 Env_Set_Cor_Timer() and Env_Cancel_Cor_Timer()

Env_Set_Cor_Timer() is called by the Core (protocol) to start a timer. Env_Cancel_Cor_Timer() is called to cancel previously scheduled timers.

3.3.2 Env_Kick_Link()

The Core calls this function on a link when a packet has been enqueued that may be eligible for transmission on the link. Env_Kick_Link() then calls Cor_Link_Available(), if the specified link is not busy. If Cor_Link_Available() returns a packet, the packet is transmitted by Env_Kick_Link().

3.3.3 Env_Enqueue_Pkt()

This function enqueues a packet onto a specified queue.

3.3.4 Env_Dequeue_Pkt()

This function dequeues a packet from the queue on which it currently is placed.

3.3.5 Env_Place_Pkt()

This function takes an unqueued packet and inserts it into the queue of a given queued packet.

3.3.6 Env_Gen_Traffic()

The TG library calls this function when the Environment should generate and send a traffic packet. Env_Gen_Traffic() then calls Cor_Packet_Arrival(), to announce the arrival of the packet to the Core.

3.3.7 Other Environment Functions

The Environment includes several other functions that are not as noteworthy as those already mentioned. We list them all here, for completeness' sake.

- Env_Time_s()
- Env_Link_Busy_Time_Rem()

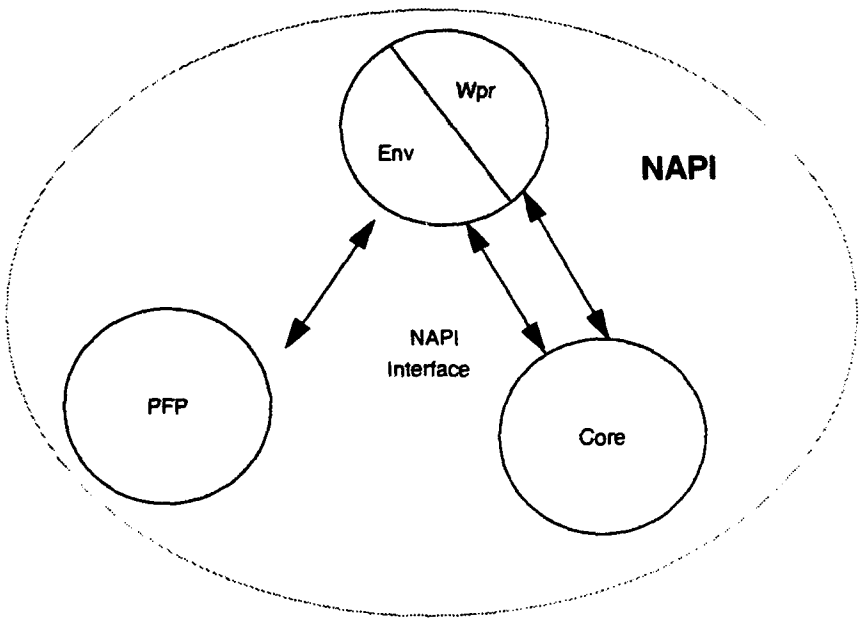


Figure 2. Overview of NAPI Components

- Env_Link_Probability()
- Env_Link_Timeout_Window_s()
- Env_Num_Links()
- Env_Link_Neighbor()
- Env_Get_Nth_Pkt()
- Env_Pkt_At_Bit_N()

  Env_Destroy_Pkt()

  Env_Pkt_Len_bits()

  Env_Pkt_Gen_Time_s()
- Env_Create_Queues()
- Env_Get_Q_Size_Bits().

3.4 CORE COMPONENT

The Core is responsible for

- Protocol-dependent algorithms, such as
  - Computing flow, link delay, and expected delay to the destination
  - Directed Acyclic Graph (DAG) management
  - Link scheduling
- Identifying the links to be activated to transmit packets
- Initiating update packet generation
- Enqueueing traffic and control packets on the appropriate queues.

Core functions are described below.

3.4.1 Cor_Epoch_Processing()

This function is called by the Environment on a periodic basis, according to the value of the input parameter epoch_interval. The Core protocol may elect to do periodic protocol processing as a result of being called by this function.

3.4.2 Cor_Link_Available()

This function is called by the Environment whenever a link is freed after a packet transmission and whenever a link is "kicked" by a call from the Core to Env_Kick_Link(). The Core should return an eligible packet if such a packet exists; otherwise, the value NULL should be returned.

3.4.3 Cor_Packet_Arrival()

The Environment calls this function to hand a packet to the Core whenever one of the following conditions exists:

- A packet has been generated locally by the node.
- A packet has arrived at the node from the network.
- A packet has not been acked and a retransmission timer has expired, indicating that the packet should be requeued for retransmission.

The Core is expected to either enqueue the packet on an appropriate queue, or destroy the packet if it has not reached its final destination.

3.4.4 Cor_Timer_Callback()

This function is called by the Environment when a Core-initiated timer has expired (via Env_Set_Cor_Timer()). The Core should perform the processing associated with the expiration of the timer.

3.4.5 Cor_Set_Update_Dest_Var(), Cor_Set_Update_Link_Var(), and Cor_Set_Update_Link_Dest_Var()

These functions are called by the Wrapper to update Core variables when an update packet has been received. While parsing through the received packet, the Wrapper calls these functions for the variables contained in the packet. One of these functions is called for every element in the variable arrays and variable matrices in the packet. The three function formats are associated with destination-oriented, link-oriented, and link/destination-oriented variables, respectively.

3.4.6 Cor_Get_Update_Dest_Var(), Cor_Get_Update_Link_Var(), and Cor_Get_Update_Link_Dest_Var()

These functions are called by the Wrapper when an update packet is being built (generated). The Core must return the current value of the indicated variable.

3.5 WRAPPER COMPONENT

The Wrapper is responsible for parsing, building, and otherwise modifying update packets. Wrapper functions are described below.

3.5.1 Wpr_Generate_Update()

This function is called by the Core when the Wrapper should generate (and send) an update packet.

3.5.2 Wpr_Parse_Update_Packet()

This function is called by the Environment to parse an update packet that has been received. The Wrapper parses through the packet, extracts the variable values, and calls one of the Cor_Set_Update functions for each value.

3.5.3 Wpr_Merge_Updates(), Wpr_Reduce_Updates(), Wpr_Combine_Updates(), Wpr_Bring_Update_Current(), and Wpr_Reduce_Old_Update()

These functions are called by the Core to manipulate the variables contained in queued update packets.

3.6 TRAFFIC GENERATOR LIBRARY

The TG Library is responsible for generating user traffic based on selected parametric traffic models.

3.7 PARAMETER FILE PARSER

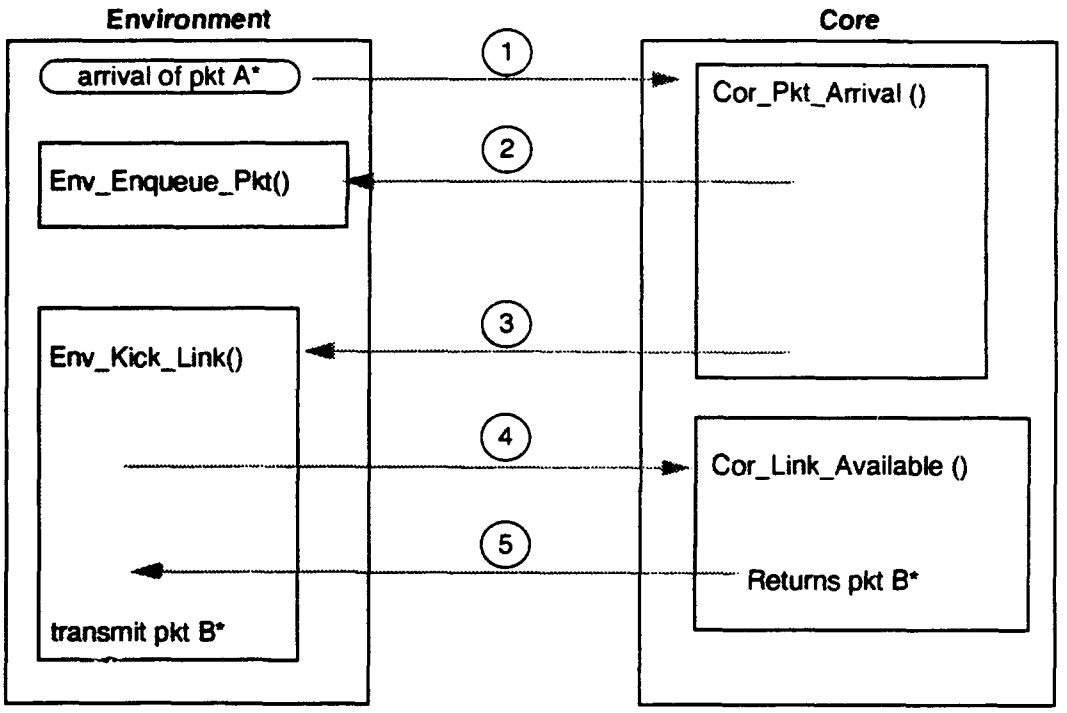
The Parameter File Parser library contains the software necessary to read the values of input parameters from the simulation parameter file.

3.8 ENVIRONMENT-CORE HANDSHAKING

We provide a few examples of event processing and show the how various functions are called in the Environment and the Core.

3.8.1 Example Packet Arrival Event

Figure 3 shows an example of interactions between the Environment and Core functions when a packet-arrival event has occurred.



*A may or may not be the same as B.

Figure 3. Packet Arrival Event Processing

3.8.2 Update Packet Generation Event

Figure 4 shows an example of interactions between the Environment and Core functions when the Core decides to generate (and send) an update packet.

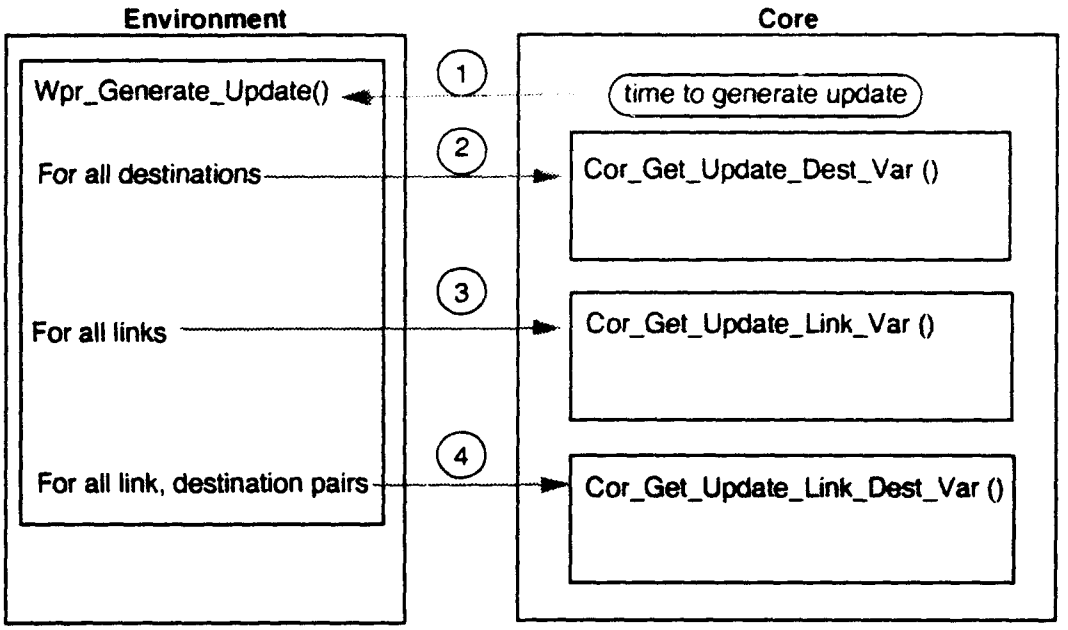


Figure 4. Generate Update Event Processing

4 NETWORK ALGORITHM PROGRAMMER'S INTERFACE FOR C++

NAPI+ is a specification for object-oriented protocol development that frees the protocol developer from much of the drudgery of a full protocol implementation. In this section, we present the background for the development of NAPI+, including the history of NAPI; we also provide a summary of the NAPI+ specification.

4.1 BACKGROUND

NAPI, the predecessor to NAPI+, was developed in order to provide an efficient mechanism for transitioning protocol software from a simple SNAPS tool implementation to a fully detailed OPNET-based simulation. The protocol developer could develop one protocol module that could be linked with either the NAPI-SNAPS or the NAPI-OPNET environment modules.

We still needed an efficient mechanism for transitioning the protocols from the simulation environment to the emulator. Prior to the development of NAPI, the Multimedia Network Emulator was built on object-oriented principles and implemented in C++. However, the NAPI specification, the associated environment modules and protocol modules, and the vendor-supplied OPNET library were written in C. C++ code and C code do not always interface with each other cleanly, especially in the case of the OPNET software package, where the function prototype specifications were not well publicized and were subject to change. In addition, the NAPI specification did not mesh well with traditional object-oriented design concepts.

4.1.1 Goals

If we had implemented STIP3 in the NAPI format (for the SNAPS and OPNET environment), we would have incurred the extra cost of porting STIP3 to the emulator. We would also have paid the heavy maintenance costs associated with synchronizing the two protocol implementations. Therefore, our main goal was to be able to implement STIP3 (and future protocols) both in a new NAPI-like simulation environment and on the emulator, while avoiding a heavy porting effort. Additionally, the NAPI-like specification either needed to be completely specified in C++, or needed to provide a model for easily interfacing C functions in the OPNET library with C++ classes.

4.1.2 Solutions

Analysis of our existing STIP1-OPNET and STIP2-NAPI/OPNET simulations showed that we were making little use of the services provided by the OPNET package. We were not using the OPNET software library to a large extent, and we rarely used the OPNET application tool. Therefore, our solution started with implementing OPNET-like services in C++. This new pseudo-OPNET package contained the small subset of OPNET functionalities that we were using in the NAPI simulations. Once we had access to these services via C++ software, the rest of the solution became obvious. We designed the new NAPI-like interface, using C++. This interface provides an object-oriented mechanism for accessing the services we had previously built into NAPI. This interface specification is now known as NAPI+.

The remaining effort consisted of developing a new environment module compliant with NAPI+. This new environment module incorporated the pseudo-OPNET software and a porting of the existing software in the NAPI-OPNET environment module.

An extra advantage of this solution is that future simulation developers will not need to purchase the OPNET package in order to modify, extend, or otherwise maintain the environment software, because the OPNET software library and the OPNET tool are not a part of the NAPI+ development environment.

4.1.3 Implementing Protocols in NAPI+

Beginning with STIP3, we were not required to implement protocols in both the simulation environment and the emulation environment. All new protocol code is written in C++ and compliant with NAPI+. Once the new protocol module has been implemented, it can be used in either the simulation or the emulation environment.

Although we had already implemented the Baseline protocol and the STIP2 protocol in the NAPI environment, we chose to reimplement them in NAPI+. The Baseline protocol was the simplest of the existing NAPI protocols at the time, so we used Baseline as the test-case protocol to shake out the bugs in the NAPI+ environment module. We reimplemented STIP2 for two reasons. First, we needed a protocol to test some of the sophisticated services of NAPI+ that are not used by Baseline. Second, NAPI+ provided a much more efficient way of minimizing the sizes of packets: the protocol developer could specify a completely different packet header for each type of packet; also, the object-oriented features provided by "variable blocks" enabled the developer to reduce the size of update packets substantially. Therefore, it would not have been meaningful to compare the STIP3-NAPI+ and STIP2-NAPI implementations.

To date, four protocols have been implemented as NAPI+ modules: Baseline, STIP2, STIP3, and OSPF.

4.2 NAPI+ SPECIFICATION

NAPI+ is a specification for an object-oriented programming environment for network protocol development. The NAPI+ environment is an implementation of the classes and functions in the NAPI+ specification and many other classes and functions that are not visible in the specification. The specification basically consists of three kinds of classes: (1) classes that provide protocol-independent services that are useful for implementing a full simulation or packet switch; (2) protocol-independent base classes from which the protocol developer must derive protocol-dependent classes; and (3) an assortment of minor functions that the protocol module must implement to bind the environment to the protocol.

This section provides an overview of the main features of the NAPI+ specification and the underlying services provided by the NAPI+ environment. We intentionally avoid duplicating all the NAPI+ details by referring the reader to Appendix D, How to Implement a Protocol in NAPI+ and the NAPI+ header (.h) files in SRI's release of the NAPI+ environment module.

4.2.1 Mandatory Protocol Classes and Functions

To use the NAPI+ simulation environment, the protocol developer must define and implement several protocol-dependent C++ classes that are derived from classes in the NAPI+ specification. Defining a required derived class consists of creating a C++ class specification and implementing

the class member functions and data structures to execute the protocol-specific behavior. In addition to implementing the required classes and functions in the protocol module, the protocol developer has complete freedom to create additional functions and classes. Figure 5 summarizes the protocol-dependent classes and functions that the protocol developer must implement.

4.2.2 Manipulating Update Data

NAPI+ provides a collection of classes that the protocol module can use to store the values of update variables. The term "update variables" is used here to mean variables that are passed in update or hello-type packets from one node to another. Several classes are provided for representing update variables as arrays and matrices in different formats. For the purposes of discussion, we will refer to objects of these classes as "variable blocks." Included in the classes are several functions that can be used to manipulate the variable data stored in the variable block object.

All variable-block classes are derived from the base class N_VB. The specific variable-block class the protocol uses depends on whether the variable represented is a matrix, an array, or a single element. In addition, different classes are provided for matrices and arrays depending on whether or not the protocol is sending an image of the entire variable block (or part of the variable).

Once the variable-block object is created, it can be appended to a list of variable blocks. The NAPI+ class N_VBList is provided for this purpose. This variable list can then in turn be used as a data element in an update packet.

4.2.3 NAPI+ Packet Queues

NAPI+ provides the packet-queue classes N_PktQueue and N_PriorityTimeQueue (which is derived from N_PktQueue). N_PktQueue is simply a linked-list class that can be used to store packets. N_PktQueue's enqueue() function simply adds the given packet to the end of the list. N_PriorityTimeQueue's enqueue() function enqueues the given packet by priority and arrival time. The protocol developer can choose to use either of these two classes, or define a new class derived from N_PktQueue.

With the exception of the function enqueue(), the same basic queue-access functions are used, regardless of whether the developer chooses to use N_PktQueue or a class derived from N_PktQueue. This is possible since the basic queue-access functions are member functions of the base class N_PktQueue.

4.2.4 Array and Matrix Classes

NAPI+ provides Array and Matrix classes, which the protocol developer can use to store the val of matrix and array data. The chief advantage of using these classes is that they conveniently provide services such as memory allocation, memory deallocation, and index range checking, by simply constructing an object of the class. These classes are

- N_ArrayInt
- N_ArrayFloat
- N_MatrixInt
- N_MatrixFloat.

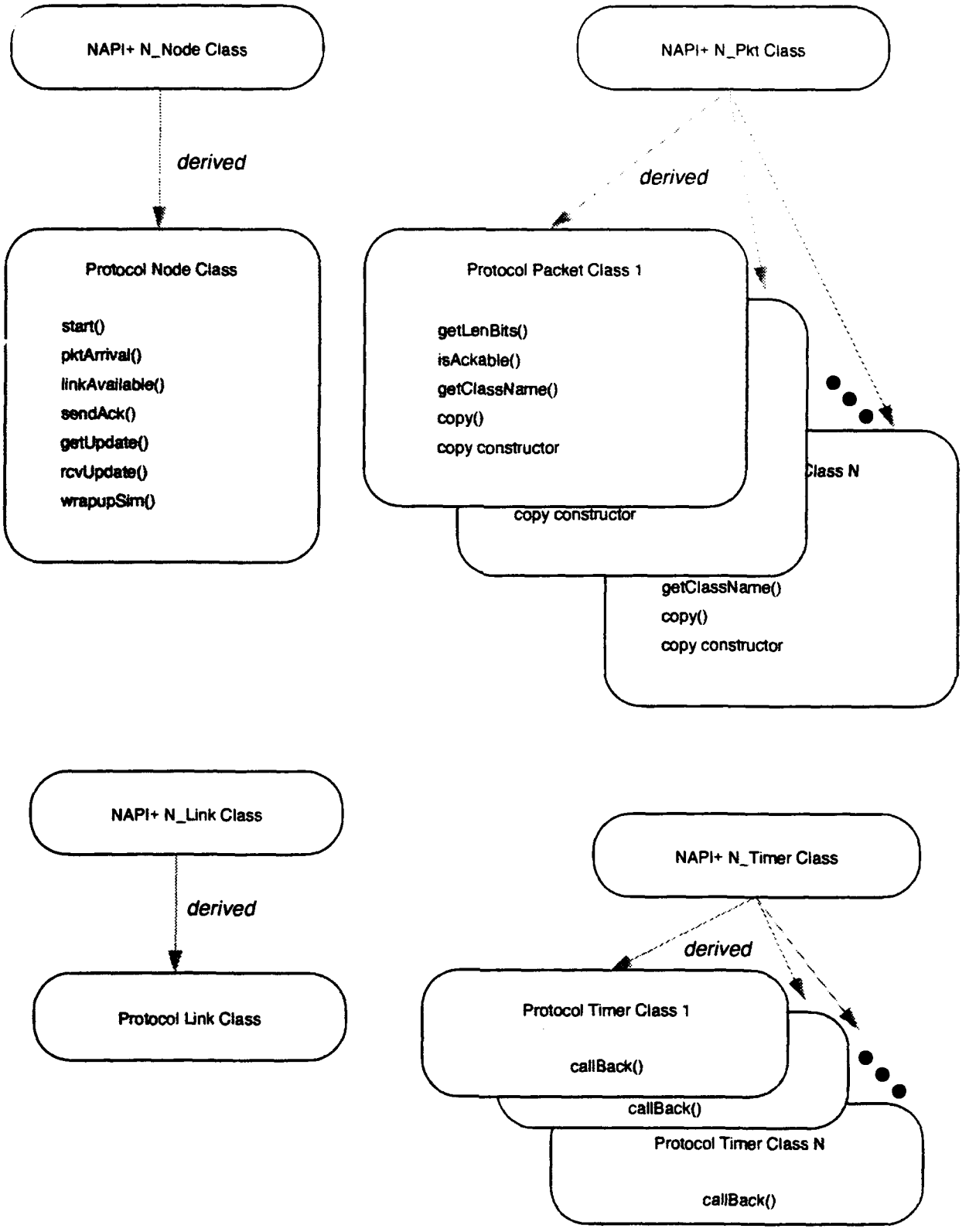


Figure 5. Mandatory Protocol Classes

4.2.5 Logging Variables

NAPI+ provides a method for the periodic automatic logging of variable values. The logged values can be examined after the simulation for the analysis of protocol behavior. To enable this feature for a given variable, the protocol developer uses one of the NAPI+ array or matrix classes specified in the section above to construct an object. The object is then added to a list of objects that is logged automatically by the environment.

5 THE STIP PROTOCOLS

Under the EDMUNDS project, SRI developed the Secure Tactical Internet Protocol (STIP) series of three increasingly efficient and robust protocols. STIP1 was the first of the series, representing an exciting new direction for survivable routing/flow control protocols for the tactical environment. Specifically, STIP1 has been developed upon a theoretically sound platform to provide high-throughput, low-delay, fair service in multimedia networks that are subject to jamming. The protocol makes use of alternate routing to select outgoing links (among all media) based on local information about dynamic link-states and distributed information about congestion information.

STIP2 improved upon STIP1 by using several new methods for better performance and increased robustness against a "protocol-smart" adversary. These improvements include a new algorithm for maintaining the directed acyclic graph (DAG); an improved flow-computation algorithm that predicts and avoids congestion; a method for spreading traffic over multiple paths for increased robustness to jamming; and an improved link-scheduling algorithm.

STIP3 improved upon STIP2 by using a faster-responding link probability estimator, an improved link scheduler that supports multiple priorities, and a flow computation algorithm that optimizes a global rather than a local objective.

In order to gauge the behavior and performance of each of the STIP protocols, we analyzed them against a baseline protocol. STIP1 was compared to a baseline protocol having three variations, which is described below. STIP2 and STIP3, on the other hand, were compared to a baseline protocol (see Section 6) that is comparable to the most sophisticated variation of the baseline protocol used in the STIP1 analysis. Our simulations of the various protocols were characterized by the number of nodes (10, 25, and 50) and by network dynamics (none; slow [2 s, 5 s, and 10 s jamming periods]; fast [0.25 s jamming period]; and subject to random, spot, and circular jamming). Approximately 25-30 experiments were designed, and approximately 300 simulations executed for each STIP protocol analysis.

This section provides summary descriptions of the STIP protocols, and of their performance as compared against a baseline protocol (Baseline).

5.1 MODEL

We assume a multihop multimedia network with point-to-point links. (Radio links can be assumed to be point to point, if time-division multiple-access (TDMA) scheduling is used to avoid collisions.) Each link is characterized parametrically by its capacity, propagation delay, and timeout for retransmissions; these parameters can vary widely for different media. The probability that each link is up can vary dynamically, depending on the jammer, mobility, or environmental conditions. High-performance packet switches are assumed that can dedicate approximately 100 MIPs to protocol processing. Refer to the documents describing the EDMUNDS Network Environment Profile* for a complete description of the general network environment of interest to the EDMUNDS project.

5.2 COMMON FEATURES

STIP1, STIP2, and STIP3 have the following common features:

- All are distributed (distance-vector) routing algorithms.
- All use distance variables $e(i,d)$ that are time smoothed and are sent to neighbors only if they change by at least some threshold.
- All use destination queues instead of link queues, so that the routing decision can be made at the last possible moment.
- All estimate the probability of success for each outgoing link based on received acks.
- All compute locally optimal flows based on local queue sizes, capacities of outgoing links, estimated probabilities of outgoing links, and expected delays from neighbors to each destination.
- All allow updates and acks to be routed to neighbors on indirect paths.

5.3 STIP1

STIP1 is based on a distributed, distance-vector algorithm. Each node receives and stores the estimated delays $e(j,d)$ sent from its neighbors. Based on these and the above link parameters, the node computes flows $f(l,d)$ and threshold values $\theta(l,d)$, which determine the assignment of packets to links. Time averaging of $e(d)$ is progressive, so that delays are averaged more the farther they are propagated. Overhead traffic is reduced via thresholding, to limit the broadcast of updated state information to times when the changes are significant.

The main metric used in the protocol is $e(i,d)$, which is roughly the expected time required for the last packet, currently at node i and destined for node d, to arrive at d. This expected delay depends on the estimated delays $e(j,d)$ sent from neighbors, the current estimated state of the outgoing links, the current number of packets at node i bound for each destination, and the routing policy. Corresponding to these short-term delays, long-term average delays are also maintained.

Now each node must compute for each destination and each outgoing link a flow $f(l,d)$ that gives the rate at which packets destined for d are transmitted on link l. For each destination d, we would like to minimize the expected delay $e(i,d)$ for the last packet in the node destined for d to reach d. Because of queuing delay, not all packets will be sent on the link for which the delay to d is minimum; thus, the flows for each destination may be spread over several links. Because the different destinations are competing for the same outgoing links, we must share the link capacity fairly. The computation of the flows $f(l,d)$ is based on the current probabilities of the outgoing links, the number of packets at node i bound for each destination, and the objective of minimizing Σ_d $e(i,d)$. While the best link for a particular destination always serves the highest-priority (closest to head of queue) packet with that destination, an inferior link may serve a packet farther back in the queue.

---

*Lee, D.S., D.A. Beyer, and R.G. Ogier. 1992a. *EDMUNDS Network Environment Profile: Benign and Adversarial Conditions, and Benchmark Scenarios (Part I)*, ITAD-8558-TR-91-23, SRI International, Menlo Park, California (May). Lee, D.S., D.A. Beyer, and R.G. Ogier. 1992b. *EDMUNDS Network Environment Profile: Benign and Adversarial Conditions, and Benchmark Scenarios (Part II)*, ITAD-8558-TR-91-23, SRI International, Menlo Park, California (July), SECRET.

Each link keeps track of the current probability that it is up. These variables are computed on the basis of the history of packet success. In future systems, messages from neighbors containing received noise levels may also be used to help compute this probability. In addition, each link tracks the average time before acks are received for transmissions, and updates an ack timeout window accordingly.

To avoid routing loops that could be caused by network dynamics, a distributed algorithm computes and updates a DAG(d) for each destination, based on the expected delays to that destination. Packets being transmitted toward destination d are limited to links in DAG(d). Each node considers only links in the current DAG(d) when computing the flows $f(l,d)$.

We now describe the queuing and retransmission rules. Each node maintains a queue corresponding to each destination. Packets are prioritized in the order of arrival: first come, first served (FCFS). When a link l selects a particular destination d for the next packet to be transmitted, the packet in queue d of the highest priority that is above the threshold $\theta(l,d)$ is selected. After a link selects and initiates the transmission of a packet, the packet is removed from the queue, and added to a list of packets awaiting acks. If no ack is received for a packet after the timeout window for the link used in the original transmission, the packet is reinserted into the appropriate destination queue, according to its original time of arrival. The packet can then be retransmitted on any appropriate link. We are not concerned with preserving end-to-end packet order, and several packets can be transmitted on a link before an ack is received for the first.

5.3.1 Comparison of STIP1 to Well-Known Algorithms

STIP1 is a distance-vector algorithm [Schwartz 1986]; that is, it belongs to the class of distributed routing algorithms in which each node i informs its neighbors of the delay from node i to every destination. The distributed Bellman-Ford shortest-path algorithm is another example of a distance-vector algorithm*. However, the Bellman-Ford algorithm is a single-path algorithm, while STIP1 is a multipath algorithm.

In STIP1, each node computes its routing strategy (flows), based on the current local queue-size and link-state information, and on time-averaged delay information received from neighbors. This combination of local dynamic and global quasi-static response sets STIP1 apart from strictly quasi-static algorithms such as Gallager's algorithm†, which do not respond to dynamic changes in local queue sizes or link states.

Although dynamic routing algorithms based on queue sizes exist‡, distributed versions of these algorithms exist only for the case of a single destination and zero link-propagation and link-transmission delays.**

---

*Schwartz, M. 1986. *Telecommunication Networks: Protocols, Modeling and Analysis.* Addition-Wesley Publishing Co., Reading, Massachusetts. Bertsekas, D., and R. Gallager. 1987. *Data Networks* Prentice-Hall, Inc., Englewood Cliffs, New Jersey.

†Gallager, R. 1977. "A minimum delay routing algorithm using distributed computation," *IEEE Trans. on Communications*, COM-25(1):73-85 (January).

‡Hajek, B., and R.G. Ogier. 1984. "Optimal dynamic routing in communication networks with continuous traffic," *Networks*, 14:475-487. Ogier, R.G. 1988. "Minimum-delay routing in continuous-time dynamic networks with piece with constant capacities," *Networks*, 18:303-318.

**Hajek, B., and R.G. Ogier. 1984. "Optimal dynamic routing in communication networks with continuous traffic," *Networks*, 14:475-487.

Two routing algorithms that respond dynamically to local queue size are delta routing* and shortest queue plus bias†. The following paragraphs discuss how STIP1 differs from these two algorithms.

Delta routing and shortest queue plus bias, like most routing algorithms, assume that packets are queued only at the links, and in first-come first-served fashion. STIP1 instead queues packets at the nodes (one queue per destination), so that the decision of which link to use for a given packet can be made at the last possible moment. STIP1's queueing method also allows the different destinations to be served in weighted round-robin fashion (for improved fairness) rather than in first-come first-served fashion for each link. In STIP1, each node i computes the optimal flow allocation so as to minimize the sum of the delays over all destinations. These flows are used as the weights in the weighted-round-robin link scheduler.

In addition, unlike previous algorithms, STIP1 uses a novel technique of taking a packet from the middle or end of the queue (rather than the head), when using alternate links of higher delay. This technique minimizes the maximum as well as the average expected packet delay.

If delta routing is to use memoriless routing, rather than source routing, then the set of paths for each source-destination pair must be selected carefully to avoid loops. Letting $e(i,d)$ denote the computed delay from node i to destination d, loop avoidance can be achieved by allowing node i to use link (i,j) for packets of destination d, only if $e(i,d) > e(j,d)$. However, this technique is restrictive and may prevent the use of any alternate path being allowed, even if one exists and is needed. (For example, there may be only one neighbor j of smaller delay.) STIP1 also avoids long-term loops by requiring $e(i,d) > e(j,d)$ when link (i,j) is used for traffic destined for d. However, in contrast to delta routing, STIP1 allows $e(i,d)$ to be increased, to allow the use of multiple paths to a destination when they are required.

While delta routing time-averages link delays, STIP1 time-averages the delays $e(i,d)$. This time averaging allows delay information to be averaged over successively longer intervals the farther updates propagate: this technique is intuitively appealing, because information from distant nodes is delayed more than information from nearby nodes. In addition, STIP1 reports a change in $e(i,d)$ only if the change is greater than some threshold ε, in order to reduce the amount of update traffic.

5.3.2 Performance Results

STIP1 represents the initial version of a radical departure from conventional methods for routing and flow control in tactical networks. STIP1 is constructed from a number of new and a few existing algorithm components to form a complete network protocol. The performance results, compared to those of the EDMUNDS Baseline protocol, were favorable. In the subsections that follow we describe the EDMUNDS Baseline protocol and summarize the results of our performance analysis of STIP1. For further details of the STIP1 algorithms and performance, please see the STIP1 document referenced below.‡

---

*Rudin, H. 1976. "On routing and 'delta routing': A taxonomy and performance comparison of techniques for packet-switched networks," *IEEE Trans. on Communications*, COM-24(1):43-59 (January).

†Fultz, G.L., and L. Kleinrock. 1971. "Adaptive routing techniques for store-and-forward computer-communication networks," in *Proc. ICC*, pp. 39.1-8.

‡Beyer, D.A., J.M. Hight, R.G. Ogier, and D.S. Lee. 1993. *Secure Tactical Internet Protocol 1 (STIP1)*, ITAD-8558-TR-93-31, SRI International, Menlo Park, California (February).

5.3.2.1 Baseline Protocol

In order to gauge the performance and behavior of STIP1, SRI constructed a simple baseline protocol (Baseline) to be subjected to the same tests. Baseline shares the same link computations and maintenance, expected delay smoothing and thresholding, and queue prioritization algorithms that were introduced during the project for STIP1. However, Baseline uses simple forwarding and alternate routing rules in place of the sophisticated flow computation and link scheduling algorithms used in STIP1. Specifically, Baseline differs from STIP1 in three major ways:

- Traffic packets are assigned primary and secondary outgoing links upon arrival at a node according to the expected delay to the destination; thus, link scheduling frames are not used.
- The expected delay to the destination is computed via the distributed Bellman-Ford algorithm, with the metric being the link delay that includes the smoothed link-queueing delay; thus, flow variables, queue thresholds, and τ one-hop delays are not used.
- Acknowledgements are transmitted on the same link as was used by the received data or control packet.

Three variations of the Baseline protocol were used whose differences lay in the link metric; they are

- Baseline1—min-hop routing (link delays were fixed at 1.0)
- Baseline2—min-delay routing (smoothed value for link queues)
- Baseline3—instantaneous min-delay routing (instantaneous value for link queues).

Recall that in performance studies done on STIP2 and STIP3, only one Baseline protocol was used. That version is most similar to the Baseline3 protocol, and is more fully discussed in Section 6.

5.3.2.2 Conclusions

The following is a list of our main findings from simulations run on STIP1 and on the various versions of our Baseline protocol:

- STIP1 and Baseline3 perform significantly better than Baseline1 in most scenarios.
- STIP1 requires an initialization phase while the DAG is being formed.
- STIP1 operates with less channel overhead and thus a higher throughput than Baseline3 (about 15% higher) and Baseline2 (about 20% higher), for simple scenarios.
- Baseline3 performs better than STIP1 in small and medium-size networks (10-25 nodes) with no dynamics.
- STIP1 outperforms Baseline3 in large (i.e., 50-node) networks with no dynamics.
- STIP1 performs significantly better under fast dynamics than under slow dynamics.
- Baseline3 performs slightly better than STIP1 in small networks under fast dynamics.
- STIP1 outperforms Baseline3 in small networks under slow dynamics.

5.4 STIP2

This section summarizes the STIP2 protocol and its performance in comparison with that of Baseline. For further details of the STIP2 algorithms and performance results, see Ogier and Lee [1993].

5.4.1 STIP2 Protocol Summary

As a result of our STIP1 simulations and analysis, we have developed a number of new methods for improving the performance of STIP1 and increasing its robustness in the presence of sophisticated jamming. These improvements have been implemented into STIP2 and are summarized below.

The STIP1 link-scheduling algorithm operates with a fixed scheduling frame per link. This method does not adequately handle burstiness, since bursts must be spread out over multiple frames. STIP2 uses an improved link scheduling algorithm that implements a leaky-bucket flow-control mechanism for each link-destination pair. This algorithm improves STIP2's responsiveness to bursts, while maintaining the integrity of long-term flows.

The local objective function used by STIP1 to compute the link flows $f(l,d)$ attempts to minimize the delay of only the packets currently queued at the node, ignoring future arrivals. For a constant stream that uses two outgoing links having different delays to the destination, this objective function causes a permanent queuing delay for the best link. This delay will be equal to the difference between the delay to the destination incurred by the use of the second-best link, and the delay when the best link is used. STIP2 uses a modified objective function that also considers the queuing delay of packets that are expected to arrive at the node in the near future. STIP2 thus predicts and avoids queue buildup by routing traffic on alternate links *before* the queue builds up.

In addition, STIP2 uses a method for spreading traffic over multiple paths, in order to improve stability and provide robustness to sophisticated jamming. The degree of spreading can be controlled by an input parameter. STIP2 also uses an improved method for maintaining the destination-oriented DAGs, including a new update message that informs neighbors when a link has been added to or removed from each DAG. This method prevents all two-hop loops and accelerates the response to link failures.

Other improvements implemented into STIP2 include the following:

- Update packets having the same source and destination, but generated at different times, are combined into a single packet for improved efficiency.
- Updates are prioritized so that less important updates are sent less frequently.
- A method is used that allows a node to detect that it is jammed, so that it can inform its neighbors of its condition. This prevents the node's neighbors from attempting to route to it along alternate paths.
- An option was added to allow or disallow acks and updates to be routed along an indirect path (instead of a direct link).

5.4.2 Performance Results

STIP2 contains a number of improvements over STIP1, our initial version of a radical departure from conventional methods for routing and flow control in tactical networks. STIP2 has an improved link-scheduling protocol, an improved objective function used to compute the flows

over each link for each destination, and a faster DAG formation and updating algorithm. Simulations showed that in general STIP2 performs better than the Baseline protocol, and in many cases is far superior.

The following lists our main findings from simulations run on STIP1 and Baseline.

- STIP2 operates with less channel overhead and thus a higher throughput than Baseline (about 6%).
- STIP2 outperforms Baseline in small and medium-size, simple networks with no dynamics (40%-55% lower mean ETE delays).
- STIP2 performs better than Baseline in small and medium-size (i.e., 10-25 node), complex networks with no dynamics (15%-30% lower mean ETE delays).
- STIP2 performs better than Baseline in large (i.e., 50 node) networks with no dynamics (25%-30% lower mean ETE delays).
- STIP2 outperforms Baseline in small and medium-size networks with slow dynamics (40%-75% lower mean ETE delays).
- STIP2 performs better than Baseline in small and medium-size networks with fast dynamics (7%-15% lower mean ETE delays).

5.5 STIP3

This section summarizes the STIP3 protocol and its comparison against Baseline. For further details of the STIP3 algorithms and performance results, see Appendix E.

5.5.1 STIP3 Protocol Summary

With the help of feedback from a large number of simulation experiments on STIP2, we developed several improvements and implemented them into STIP3. The most important of these improvements are summarized below.

STIP1 and 2 use an algorithm for estimating a link's probability $p(l)$ of success, based on received acks. This method requires averaging over an interval at least two times the timeout period, which results in a response time of about 1 second. STIP3 uses a new algorithm that responds faster to links coming up. The STIP3 algorithm exploits the following inherent asymmetry: if an ack has been received for a transmission, then we know the transmission was successful; but if an ack has not yet been received, we do not know whether the transmission was successful. Based on this asymmetry, STIP3 computes two different estimates of $p(l)$, one of which is a lower bound, and then sets $p(l)$ to the maximum of these estimates.

In addition, we found in simulations that the optimal smoothing constant for $p(l)$ (i.e., the degree to which $p(l)$ is time averaged) depends greatly on the rate of link dynamics. Therefore, STIP3 provides the option of automatically adjusting this smoothing constant according to the measured link dynamics.

From simulations we discovered two drawbacks of the STIP2 link scheduling algorithm:

- Traffic of a given destination was not guaranteed to be divided among outgoing links in proportion to the flows.
- Traffic with large bursts was not treated fairly in that the computed flows are based on time-smoothed measurements of the incoming traffic.

STIP3 uses an improved link-scheduling algorithm that overcomes both drawbacks, while preserving the following benefits:

- Destination queues are still used, so that the routing decision can be made at the last possible moment.
- A link l that has higher delay than the primary link for some destination d may select a packet from the middle or tail of the queue for d, depending on the computed variables $\theta(l,d)$, in order to minimize maximum packet delay.

In addition, the STIP3 link scheduling algorithm supports multiple priorities, and includes a method that allows the limiting of high priority traffic (if desired) in order to prevent lower priority traffic from being squeezed out.

While STIP2 computes flows by minimizing a local objective function, STIP3 attempts to minimize a global objective function representing the average network delay. This is accomplished by letting $e(i,d)$ denote the marginal cost (i.e., the amount the global objective increases, due to a small increase in local traffic) instead of average delay. Since the new method of computing flows provides better performance than the STIP2 method in some scenarios but not others, STIP3 also provides the option of computing flows as in STIP2. In addition, STIP3 provides the option of using a simple flow-control mechanism that rejects packets entering the network if $e(i,d)$ exceeds some threshold.

5.5.2 Performance Results

STIP3 contains a number of improvements over STIP2 and STIP1. These include new link-state-estimation, link-scheduling, and flow-computation algorithms. Simulations showed that STIP3 performs better than Baseline in most experiments, and in many cases is far superior.

The following lists our main findings from simulations run on STIP3 and on the Baseline protocol.

- For the same end-to-end delay, STIP3 achieves up to twice as much throughput as Baseline.
- STIP3 and Baseline are fair under bursty traffic conditions.
- Baseline was found to have as much as 6.7 times the mean end-to-end delay and as much as 3.7 times the response time of STIP3.
- For STIP3, old flows generally performed better than new flows in simple topologies with low connectivity and little dynamics.
- For STIP3, acknowledgments and updates should be sent over indirect links in scenarios with slow, random jamming in small networks (10 nodes).
- STIP3 generally outperforms Baseline in small, simple networks with no dynamics (up to 45% lower mean ETE delays).
- STIP3 performs better than Baseline in medium-size and large random networks with no dynamics (5%-25% lower mean ETE delays).
- STIP3 outperforms Baseline in small and medium-size networks with slow dynamics (15%-85% lower ETE delays).
- STIP3 outperforms Baseline in large networks with fast dynamics (60%-75% lower ETE delays, and 75%-200% higher reliability).

- Baseline reliability is at times much lower than that of STIP3, especially in large networks with "heavy" traffic or high dynamics.
- Additional tests need to be performed before definitive conclusions can be reached regarding STIP3 and Baseline performance in small, random networks with no dynamics, small and medium-size networks with fast dynamics, and large networks with slow dynamics.

6 REVIEW OF EDMUNDS BASELINE PROTOCOL

6.1 BASELINE ALGORITHMS

The Baseline protocol is based on a distributed, distance vector algorithm, where the expected delay to the destination is as expressed below:

$e(d) = min_l (D_q(l) + e(j,d) + v(l))$

where

- $e(d)$ = expected delay to destination d
- $v(l)$ = link delay over link l
- $e(j,d)$ = j's expected delay to d (j is a neighbor on link l)
- $D_q(l)$ = queuing delay for link l.

Packets are always enqueued onto the best link's queue. The best link is defined to be the link that provides the minimum expected delay to the destination, as shown in the equation above.

6.2 QUEUES

Each node maintains a queue $q(l)$, for each link l.

6.3 LINK PROBABILITIES

The estimation of link probabilities is exactly the same as for STIP1 and STIP2.* In the older NAPI simulations, the link probability depended upon whether the SNAPS or the OPNET environment module was used. The OPNET/NAPI environment used the STIP1/STIP2 link estimator, whereas the SNAPS/NAPI environment used the link estimator of the Simple Algorithm Simulation Tool.

6.4 LINK DELAYS

The link delay $v(l)$ is computed identically for Baseline as for STIP1 and STIP2.

6.5 RETRANSMISSIONS

There is no limit on the number of retransmissions for a given packet. At retransmission time, the packet is always enqueued onto the best link's queue. Earlier versions of the Baseline protocol limited the number of retransmissions.

6.6 ACK PACKETS

The Baseline protocol always uses the direct link to a neighbor for sending acks. That is, acks for packets received from node i by node j are sent over a direct link from j to i. This procedure is different from those in STIP1, STIP2, and STIP3, where updates may or may not use indirect links.

---

*Beyer, D.A., J.M. Hight, R.G. Ogier, and D.S. Lee. 1993. *Secure Tactical Internet Protocol 1 (STIP1)*, ITAD-8558-TR-93-31, SRI International, Menlo Park, California (February).

6.7 UPDATE PACKETS

Update packets are always sent to neighbors over direct links. This procedure is different from those in STIP1, STIP2, or STIP3, where updates may or may not use indirect links.

6.8 DROPPING PACKETS

Traffic packets are dropped if the node has no good link to the destination of the packet. This is different from that in STIP3, which has destination queues; also, in STIP3 packets are dropped only when a parameterized lifetime expires.

7 OSPF

The Open Shortest Path First (OSPF) protocol was implemented for simulation in the NAPI+ environment, for the purpose of comparing STIP3 to a well known algorithm, and to further test the NAPI+ environment. OSPF provides many features to satisfy the requirements of exchanging routing data between autonomous systems over a variety of physical network types. Most of these features are not of concern for the comparison to STIP3. This section describes our implementation of OSPF and how it differs from OSPF as described by the Internet Engineering Task Force Network Working Group, in an Internet draft document entitled *OSPF Version 2*, and dated March 1993. In addition, we describe our initial comparative performance analysis of STIP3 against OSPF.

7.1 IMPLEMENTATION FOR THE NAPI+ NETWORK EMULATOR

We have implemented only a subset of the features described in the OSPF specification that we felt were applicable to the comparison of OSPF with STIP3. In particular, we implemented (1) OSPF packet types and Link State Advertisement (LSA) data structures; (2) hello, adjacency establishment, and link state exchange protocols; and (3) the routing table calculation.

The STIP3 protocol is designed for networks with point-to-point links and network dynamics caused primarily by jamming. The network routing area consists of a single autonomous system of homogeneous nodes; there are no designated routers, and no externally derived routing information from other autonomous systems or configuration data. Thus, a number of features supported by OSPF were not applicable to our comparison with STIP3. Briefly, the following features supported by the OSPF specification were not implemented:

- IP subnetting
- Stub areas
- Interarea routing
- Routing to external autonomous systems
- TOS routing
- Designated routers and backup routers
- Interface types other than point to point
- Link state advertisements other than router links
- Authentication.

In addition, OSPF packets (as well as traffic packets) were implemented not "on top of IP" but as raw NAPI+ emulation packet types, with node addresses (i.e., router IDs) being integer node numbers assigned by the NAPI+ environment rather than 32-bit IP addresses.

Below, we further describe specific features of OSPF that we did or did not implement, and the additions and modifications that were needed to make possible a fair comparison of OSPF to STIP3.

Equal-Cost Multipath. As described in the specification, when multiple equal-cost routes exist for a destination, all of them are computed. In our implementation, equal routes to a given destination are assigned in a round-robin fashion when a traffic packet is being enqueued for transmission on the network. Route calculation is based upon Dijkstra's algorithm.

Interface State Machine. Since no loopback test mode or designated router procedure exists for the point-to-point interfaces used for the simulation, the interface is always in the "point-to-point" state. Therefore, the interface state machine has not been implemented.

Neighbor State Machine. The neighbor state machine has been implemented as described in the specification. Since there are no designated routers, however, the "2-way" state is eliminated. The events AdjOK?, KillNbr, and LLDown are not applicable.

Packet Types. As in the specification, all five OSPF packet types are used. In addition, we introduced the packet-type traffic ack that is used to ack traffic packets on a hop-by-hop basis.

The OSPF protocol packets, and traffic and traffic-ack packets, are formatted directly over the basic simulation packet, instead of over IP as specified by OSPF. Since we did not anticipate using large packets in our experiments, no provision was made to check for fragmentation of packets.

As encouraged by the specification, OSPF packets are given priority over traffic packets. However, traffic-ack packets are currently given the same priority as OSPF packets. This was done so that the OSPF would be comparable with STIP3, where all control packets have priority over traffic packets.

Many of the fields in the OSPF packets were not applicable to our comparative analysis, as described below. Even where a field is not used, the packet still includes the full width of the field. We included unused fields so that we could correctly characterize the full transmit-delay of a given packet. Rather than cut away all the unused fields, we chose to stick closely to the OSPF specification.

Capability Fields. The E-bit and T-bit option/capability fields, which indicate external routing capability, and TOS capability in link state advertisements and OSPF packets, are the same for all nodes in the emulation; therefore they were not assigned or checked.

OSPF Packet Header. The version number, area identification, checksum, autype, and authentication fields were not assigned or checked.

Hello Packet. The network_mask, hello_interval, options, rtr_pri, router_dead_interval, designated_router, and backup_designated_router fields are not assigned or checked. Since each interface on the emulation's point-to-point network will have only one neighbor, the hello packet has a single neighbor field, resulting in fixed-length hello packets of 48 bytes.

Database Description Packet. The option/capability field is not assigned or checked.

Link State Request Packet. In the repeatable list of requested link state advertisements, the link state type field is not assigned or checked, since only "router links" types are used.

Link State Update Packet. This packet is implemented as described in the specification, except for the modifications to the link state advertisement described below.

Link State Acknowledgment Packet. This packet is implemented as described in the specification, except for the modifications to the link state advertisement header described below.

Link State Advertisement. Of the five link state types defined in the specification, the only one used in the simulation is type 1, router links, because only point-to-point links and internal area routes are of interest. The option/capability fields (v, e, and b) are not assigned or checked. In the repeatable fields section, the interface type field is not used, and an output cost metric for only a single type of service is given.

Link State Advertisement Header. The option/capability field is not assigned or checked, nor are the ls_type and ls_checksum fields. Since we only use link-state-type router links, and for this type the ls_type field is identical to the advertising_router field, we only use the latter field.

Architectural Constants. Of the seven architectural constants, five are used. CheckAge is not used, because we assume that memory-corruption-producing checksum errors do not occur. DefaultDestination is not used, because external routing information for other autonomous systems is also not used.

7.2 INITIAL PERFORMANCE ANALYSIS OF STIP3 WITH OSPF

A two-step process was used to analyze the performance of OSPF against STIP3. The initial step was to take a rough cut at tuning the OSPF parameters. The OSPF Version 2 specification recommends default values; however, we assume these values are more appropriate for static networks unlike the topologies STIP3 is designed to handle. A simple 10-node ring topology with two disjoint five-hop paths from the source to the destination (i.e., two paths) and alternate jamming (5 s) between the last links on each path was used to determine values for the following OSPF parameters:

- Min_LS_Interval—minimum interval to accept a changed link state advertisement (LSA)
- RXMT_Interval—retransmit interval for LSA exchange protocol
- Hello_Interval—interval for sending hello packets to good neighbors
- Poll_Interval—interval for sending hello packets to bad neighbors.

Four simulations were run with various settings for these parameters, including the OSPF recommended values. Our results show that for this topology and these parameters, the values 1.0, 1.0, 3.0, and 3.0 (seconds), respectively, gave reasonable results compared to STIP3. Using these values, we next did an initial comparison of STIP3 with our implementation of OSPF on three experiments previously performed with STIP3.

Unlike STIP3, our implementation of OSPF does not utilize the concept of DAGs. With STIP3, packets being transmitted for a node toward destination d are limited to links in DAG(d), thus providing readily available alternate routes. With OSPF, multipath provides for a form of alternate routing; however, all multipath routes are of equal cost (i.e., an equal number of hops to the destination). Packets are transmitted to destination d from a node in a round-robin fashion over these multiple paths. Thus, if there is only one shortest path to the destination, all packets will be transmitted and queued for that path, even though there are longer paths that are not busy; if the offer rate exceeds the capacity of the outgoing link of this one shortest path, packets are queued for the link as long as the path is good. The effects of this multipath routing were very evident in our experiments: the performance of OSPF was inferior to that of STIP3 in both static and dynamic scenarios, depending upon network topologies and traffic patterns.

Three experiments were performed using the 10-node two-path topology described above, a 10-node random topology, and a 25-node random topology. Each topology was run with and without dynamics. In static networks, if the scenario was such that the multiple path(s) could sustain the traffic offer rate, STIP3 and OSPF were comparable. However, if the multiple path(s) could not sustain the traffic offer rate, STIP3 outperformed OSPF; in one example, STIP3 showed a 95% lower mean ETE delay than that of OSPF.

In dynamic networks, the same effects of the OSPF multipath implementation are evident. In addition, jamming causes building and draining of queues, especially when jamming is slow, due in part to the time it takes to update link state information. We ran both the 10-node and 25-node random topologies with slow jamming; in both these cases, STIP3 mean ETE delays were 91% and 93% lower than OSPF, respectively.

7.3 CONCLUSIO[illegible]

We have perf[illegible] only a limited number of simulations of OSPF and comparisons with STIP3. Rather than using the OSPF default parameter settings, we selected settings by experimentation (and some ingenuity), which gave better results (than the default settings) on scenarios for which STIP3 was designed. These settings were then used to test a small number of scenarios. Additional work is needed, however, in comparing STIP3 to OSPF.

Since OSPF was not designed for dynamic networks, a common ground should be found on which to make a comparison between OSPF and STIP3. One approach is to fine-tune OSPF parameter settings for STIP3-type networks. A second possibility is to take a static network for which OSPF and its default parameters were designed, adjust STIP3 parameters for this network, and then add dynamics. Either of these approaches would provide a more meaningful comparison between these two protocols.

8 EMULATOR DEVELOPMENT

The following describes the design and development of the Simple Emulator. The objective of this software effort was to develop a multimedia network emulator that would be portable to a variety of hardware platforms that use the standard Mach OS.

The NAPI+ simulation environment software provided the foundation for the emulator. In addition to porting the simulation environment to Mach, we created the Mach Toolkit and developed emulation-specific software. Below we describe the Mach Toolkit and the architecture of the new Simple Emulator.

8.1 MACH TOOLKIT

SRI created the Mach Toolkit to provide support for the development of applications on the Mach Operating System. The Mach Toolkit makes Mach applications more efficient by providing a framework for program development. The toolkit automatically provides and handles the details of many Mach-unique mechanisms that applications may need, such as the creation of tasks and threads, and the allocation of ports and messages for interprocess communication.

8.1.1 Overview

The Toolkit is object oriented and coded in C++; it consists of a number of C++ classes that provide an interface to some of the Mach mechanisms. These main classes include

- A class to represent each Mach *task*
- A class to represent each Mach *thread*
- A class to handle and coordinate output from multiple threads that share the same output device
- A class to represent Mach *ports*
- A class to represent Mach *messages*.

The constructors for these various objects handle the set-up and initialization details needed for using the associated Mach service. Initialization of each of the represented Mach services is transparent to the application. The application gains access to the services via the *member functions* of each class; these functions are accessed in lieu of interfacing with the system calls provided by the Mach kernel.

8.1.2 Application Development

Developing a Mach application using the standard system calls and data structures is certainly possible. For example, a generic Mach program might begin with the main task, which consists of one thread. From this point on, child tasks can be created; and within each task, a number of threads may be spawned. Associated with each task and thread creation are a number of details. In fact, the programmer must be aware of and manage many details, to effectively make use of the multitasking/multithreading environment and the interprocess communication between threads and tasks. The toolkit provides most of the basics for a complete environment, thereby removing the burden from the application programmer of implementing all the specific details necessary for using these Mach services.

When tasks and threads are created, it is often desirable that they know about each other, can communicate with each other, and relate in other ways as well. Child tasks must register themselves with parent tasks. Tasks must know the ports of the other tasks with which they must communicate. To send messages between tasks, a Mach message structure must be used. All of these actions require some specific, detailed software that is somewhat difficult to write. The Mach Toolkit provides much of the software to handle these actions. The toolkit provides the mechanisms to register tasks and threads as they are created, provides the automatic creation and management of certain default ports, and provides a convenient way to send messages between tasks.

Note that Mach provides system calls to library functions that also attempt to give the programmer the needed tools to ease the development of a sophisticated application. However, the toolkit attempts to abstract the tools to a higher level, by defining classes for objects that will represent *tasks*, *threads*, *ports*, and *messages*. When any of these items or services are required, the programmer instantiates an object of the corresponding class. The associated object, via the constructor and member functions, handles many of the details for the programmer.

Specifically, child tasks are created by creating a MachTask object. Threads are spawned by creating a MachThread object. The toolkit also has a MachPort class and a MachMsg class. These are the main classes provided by the toolkit, in addition to others that make application development easier. The basic concept of the Mach Toolkit is that the classes it provides can be instantiated into objects to provide the major resources needed by an application, without requiring the programmer to implement all of the details.

8.1.3 MachTask Class

Mach Task, the primary class of the Mach Toolkit, is responsible for managing and tracking the tasks in the application. Each task is assigned an associated MachTask object.

The MachTask object associated with a given task will keep track of all the threads running in the task as well as all the child tasks of that task. This class also contains information concerning the parent task.

8.1.3.1 Task Creation

A MachTask object is instantiated for the main task and all the child tasks of the application. The constructor for the MachTask class initializes everything needed by a task. The first time the constructor is invoked it does not fork a new task. Subsequent creation of MachTask objects causes a child task to be forked, and the new MachTask object is associated with this new child. The parent's MachTask object is updated accordingly with information concerning the child.

Once the new task has been created, it must be detached by a call to the member function MachTask::detach(). The "detach" member function also handles registration of the child task with the parent, and calls the entry point of the main thread of the child task. In the case of the main task, the main thread of the task is still called; but there is no parent task, so the registration process is bypassed.

8.1.3.2 Task and Thread Tables

Every MachTask object contains a table of all the child tasks and child threads that it has created or spawned. The tables consists of MachThread and MachTask objects that contain information relating to the associated child tasks and threads.

8.1.3.3 Default Ports

Every task created via the MachTask class mechanism has two default ports that are automatically allocated as MachPort objects: (1) a command port to receive messages, and (2) a registration port dedicated to receiving registration information from child tasks.

Any task can easily obtain the MachPort objects corresponding to any of its child tasks or its parent task. The MachPort class provides member functions for sending messages to the associated port.

8.1 MachThread class

The toolkit provides a MachThread class that will spawn a Mach thread. The constructor initializes any required Mach OS data, registers the thread with the parent task, and blocks execution of the thread until instructed to continue by the parent thread. The MachThread class contains a pointer to the "main" function for the thread; this function is called when the thread is released by the parent thread.

The MachThread object provides the mechanisms for easy thread-flow control, for termination of the thread, and for obtaining relevant information and statistics from the thread.

8.1.4.1 Critical Code Segments

The MachLock class protects a thread from being suspended or terminated during critical sections of the code, or while shared resources are locked.

8.1.4.2 MachThread Member Functions

The following are some of the main functions provided by the MachThread class:

- suspend(): suspends the thread
- resume(): resumes execution of the thread
- terminate(): terminates the thread and unregisters it with the parent task.
- print(): prints relevant information about the thread
- myBeginNoSuspend(): protects thread from being suspend after call to this function
- myEndNoSuspend(): allows thread to be suspended after this call.

8.1.5 MachPort Class

The MachPort class is associated with Mach ports. The constructor of the MachPort class creates a new port or is used to look up and represent an existing port. A thread can easily send messages to a remote port by using an associated MachPort object. Likewise, the MachPort object for an associated local port is used to receive messages. When a thread sends a message, a reply port can specify another reply port, as appropriate.

The MachPort constructor allows the option of registering the port with the Mach Environment Manager.

The Mach Toolkit also provides timing capability via the TimerPort class. This class represents a port that a thread uses for timing purposes. The timeout is accomplished by an attempted receive operation on an otherwise unused port, using a specified timeout.

8.1.6 MachMsg Object

The MachMsg class provides a convenient interface for using the Interprocess Communication facilities of Mach.

There are two constructors, one for receiving a message, and one for sending a message.

- Receiving a Message: The constructor for a receive message allocates a buffer large enough to hold the specified size of the data and the Mach message header. The information necessary to receive a message is placed in the header. The MachMsg object is then used by the MachPort receive() function to receive a message.
- Sending a Message: The application simply passes the data buffer size to the constructor, and the Mach message header is automatically allocated and initialized. Once constructed, a MachMsg object can be passed as an argument to the MachPort send() function.

8.1.7 MachLock Class

The MachLock class provides locking of shared resources so that only one thread may have access to the resource at any one time. Using member functions of the MachLock class, the thread locks and unlocks the resource. Any other thread attempting to lock the resource is blocked until the resource is unlocked.

8.1.8 SharedOut Class

This class is used to write output to a C++ output stream that is shared by multiple threads. The toolkit provides classes derived from the SharedOut class, including:

- SharedOutFile: customized for writing to a file
- SharedOutQFiles: customized for writing to two files that are alternately used as one or the other becomes full.

8.1.9 Current Status

We used the Mach Toolkit primarily for the development of the Simple Emulator. However, the toolkit stands on its own and is useful for incorporation into other multitasking/multithreaded applications.

8.2 SIMPLE EMULATOR

The Simple Emulator was developed to provide a demonstration that the EDMUNDS/STIP protocols could be implemented on a multiprocessor packet switch. The concept is that many threads can run concurrently on separate processors, each thread performing the separate algorithms and jobs that a protocol may require. A protocol emulation designed to support this type of concurrency would be very efficient when ported to an actual multiprocessor packet switch.

The Simple Emulator demonstrated how well the protocols developed on EDMUNDS would take advantage of a multiprocessor environment, illuminated the problems associated with developing protocols to effectively take advantage of multiprocessors, and provided experience for future efforts. The approach was to implement a multitasking/ multithreaded emulator on an Encore Multimax running the Mach OS.

8.2.1 Overview

Relying heavily on an object-oriented design, the emulator was implemented in C++ on an Encore multiprocessor system running the Mach OS. The design of the emulator was influenced by the project's preexisting simulation development environment. Before the emulator was designed, protocols were being designed, implemented, and then simulated in the NAPI+ simulation environment on a SPARC workstation running UNIX. Many tools already existed to support the simulation process and the analysis of the results. Some conventions were also in place that allowed easy integration of the protocol with the simulator. The existing infrastructure for simulating the protocols was factored heavily into the design of the emulator. The most important goal was to make the porting of the protocol from the simulator to the emulator as simple, efficient, and convenient as possible.

The resulting emulator is a network emulation in which the network is composed of Node objects and each Node is an independent entity running its own copy of the protocol.

The main components of the emulator are

- Event Server: This object contains the main resources of the emulator, including the event list, a table of references to the network Nodes, and the emulation time. Access to the Event Server (a resource shared by multiple Mach threads) is protected through locking.
- Node Threads: Each Node is implemented as a Node Thread object. The Node Thread polls the Event Server for events to be processed by the Node.
- Node Event Thread: A Node Event Thread object is assigned to each thread that is spawned by the Node Thread to process an event.
- Clock Thread: This object manages the thread that is responsible for instructing the Event Server to advance the emulation time.

8.2.2 Event Server

At the core of the emulator is the Event Server. The Event Server is responsible for queueing and managing all the events that are scheduled to be delivered to nodes at specific times. The Event Server is accessible by all the Node Threads as well as the Clock Thread. Therefore, access must be regulated. The Event Server object contains a "lock" that is used to coordinate access to the Event Server. The Event Server lock assures that the integrity of the Event Server data will be maintained, while allowing the Event Server to be shared by multiple threads.

The main component of the Event Server is the Event List (see Figure 6). The Event List is a queue of all events that have been scheduled. Events are scheduled by the Nodes. Every event object contains a time to be delivered, and a handle for the Node at which the event is targeted. Typical events include packet-arrival events, link-free events, retransmission events, and events scheduled by the protocol-specific software being emulated.

The Event Server keeps a table of handles for all the Nodes in the network, and information on the current status of the Nodes. This is necessary, since the Event Server must know when Nodes are idle or are busy processing events. If any Node is busy processing an event, the Clock Thread is blocked from updating the official emulation time maintained by the Event Server. Emulation time is advanced only when all the Nodes indicate the idle state.

Emulation time is advanced one tick at a time (a tick is a fixed interval of time). The tick size is set once during initialization; the parameters of the network being emulated usually dictate the size of the tick. The Clock Thread advances the time only if both of the following two conditions are met:

- No events on the event list are still pending processing for the current time.
- All Nodes are idle (no Nodes are currently processing events).

8.2.3 Clock Thread

As previously stated, the Clock Thread is responsible for updating the Event Server time. It advances the time when all events pending for the current time slot are removed from the Event List and all the Nodes are idle. Since the official emulation time is advanced incrementally by tick intervals, a pending event would be an event scheduled for the current time or some time during

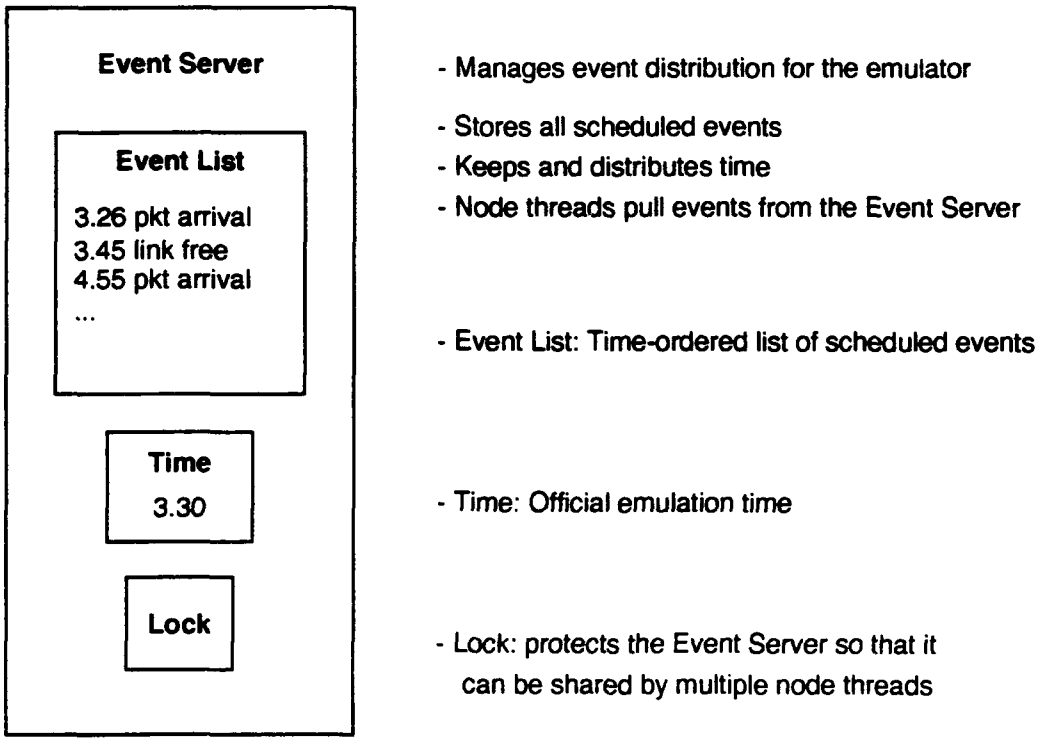


Figure 6. Emulator Event Server

the interval between the current time and the previous tick time. When the time is to be advanced, the Clock Thread instructs the Event Server to update the master clock by adding one tick interval to the time.

8.2.4 Node Thread and Node Event Thread

Every Node in the network is represented by a Node Thread. Each Node Thread executes the emulated protocol. Upon creation, every Node Thread registers with the Event Server and initializes its status to idle. Figure 7 illustrates the relationships between the Event Server, the Clock Thread, and the Node Threads.

A Node Thread polls the Event Server for events. If any events are pending for the current time, the Event Server removes the event from the Event List and returns the events to the Node Thread. The Node Thread is then responsible for processing the event.

To process the event, the Node Thread usually begins by spawning a new thread (see Figure 8). A Node Event Thread object is created to spawn the Mach thread and handle the processing of the event. Once the surrogate thread has been spawned, the Node Thread continues to poll the Event Server for any pending events. If more events are found, the events are passed to the Node Thread in the same fashion. Upon completion of processing of a specific event, the associated Node Event Thread notifies the Node Thread that the event processing is complete, and terminates the execution of the event's thread.

Whenever the Node Thread receives an event to process, it notifies the Event Server that the Node is busy (not idle). When the Node is no longer processing any events (all the Node Event Threads have terminated), the Node Event Thread notifies the Event Server that the Node is once again idle. When all Nodes are idle, The Clock Thread instructs the Event Server to advance the emulation time.

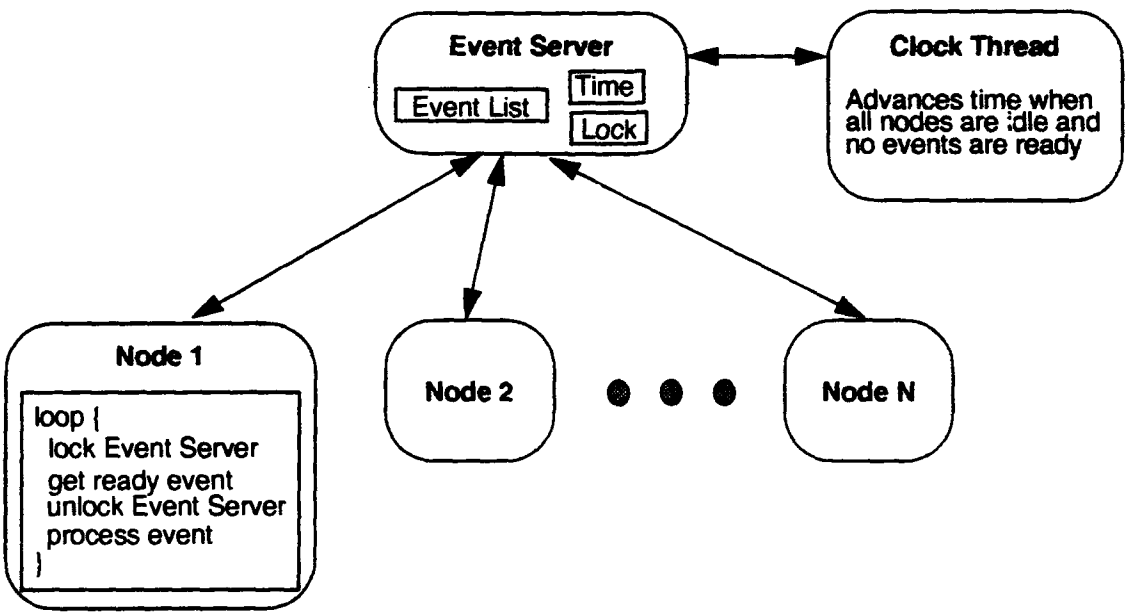


Figure 7. Emulator Architecture

8.2.5 Status

8.2.5.1 Operational Modes

To facilitate testing of the protocols in a multithreaded environment (as opposed to the single thread of the simulation environment), a number of operational modes have been built into the emulator.

Two of these operational modes, the test modes, are used to test the operation of the emulator. The other mode is the normal, or true emulator mode. The test modes are useful for debugging and may be used to analyze certain aspects of protocol behavior. The test modes are provided primarily because the protocols under analysis have been ported from a single threaded simulation environment on a different hardware platform and operating system. We can use the test modes to answer any questions about the integrity of the porting of the protocol code from the simulator to the emulator. For example, we can use the single event mode to disable effectively all concurrency within a Node; we should then see the same results that the simulation environment gives us.

The operational modes of the Simple Emulator are

- Single Node: Only one node at a time is allowed to process events. Other nodes must wait for that node to finish processing before they can access the Event List for pending events. The node may process multiple events concurrently.

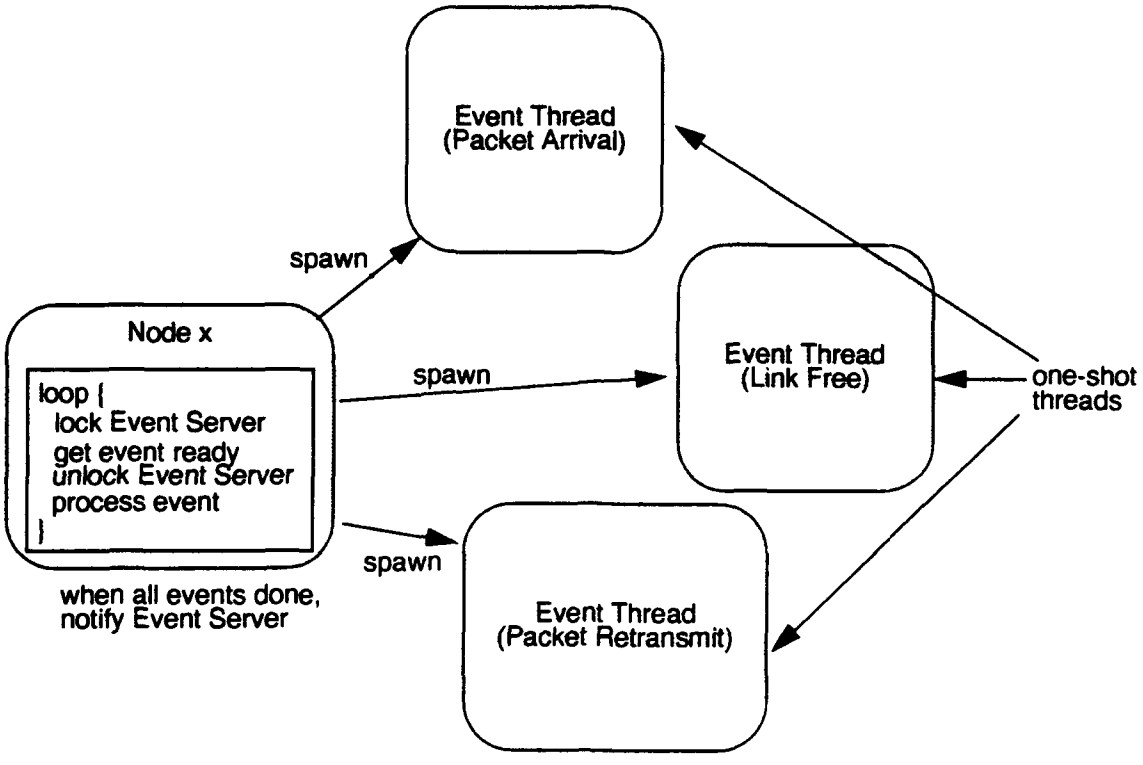


Figure 8. Spawning Node Event Threads

- Single Event: Each Node is allowed to be process only one event at a time. This restriction effectively removes any concurrency from the emulation of the protocol. A Node cannot remove a second event from the Event Sever until processing of the first event is finished. More than one Node may process single events concurrently.
- Normal: This mode is the true emulation mode. All Nodes remove and concurrently process all their events for the current time slot.

8.2.5.2 Enhancements

In the current emulator implementation, nodes are implemented as threads. In the future, it would be more realistic to create task objects to represent the nodes, thus providing independent (unshared) memory spaces for each Node. The main effort in using node tasks would be in adding a mechanism for moving packets of data between nodes. Because we currently represent nodes with threads, we can and do pass packet objects by pointer reference. We have avoided the packing of bits and bytes from packet objects into the bit-array format necessary for actual transmission on the medium ("buffer packing"). The avoidance of buffer packing does not affect what we are demonstrating (intranode concurrency). However, buffer packing is necessary, if we are to implement node tasks and create a more realistic emulation.

Another recommendation is a structural change in the way Nodes obtain events. Instead of constantly polling the Event Server for events, Nodes could wait for a signal from the Event Server. This change would make the emulation more efficient (run faster), since constant polling consumes significant amounts of CPU time.

A third enhancement would allow the emulator more flexibility in the types of entities that have access to the Event List of the Event Server. Currently, only Nodes can use the Event List resource. Future versions of the emulator may require other types of threads to use the Event List. For example, the current emulator assumes a static network with no dynamics. To incorporate dynamics, events affecting network topology would need to be scheduled. Such events would be handled by an entity other than a Node and its associated Node Threads.

Another enhancement would be to reorganize the NAPI+ interface to allow for the layering of different protocols in the emulation, including lower-layer framing and upper-layer protocols such as IP and TCP. This option would be coordinated with the NAPI+ simulation environment, where it would be useful to simulate multiple layered protocols as well.

Multiprocessor workstations are becoming faster and cheaper by leaps and bounds, and it is not at all clear that Mach will be supported on all of these hardware platforms. A fifth option would be to port the emulator to a more economical and better-supported operating system such as Sun Solaris or Windows NT. This option would make the emulator less costly (eliminating maintenance cost for the Encore Multimax), and the emulator would be accessible to more experimenters.

Last, but not least, the current NAPI+ protocols are not implemented so as to achieve efficient multithreading/multiprocessing. More work needs to be done to allow protocols to make better use of multiprocessor packet switches.

9 CONCLUSION

In this report we have presented the results of our research under the EDMUNDS project. In the following section, we summarize lessons learned and our main accomplishments, and we present ideas for further EDMUNDS-related research.

9.1 LESSONS LEARNED

We derived the following solutions from lessons learned during EDMUNDS research:

- **Establish an Algorithm-to-Protocol Transition Mechanism.** It is crucial to establish an efficient mechanism for transitioning ideas from algorithm development to the software implementation of a full protocol package. Two solutions are possible.
  - If algorithm development and protocol development are handled by separate individuals, then the algorithm developer must present clear, concise descriptions of the protocol, including pseudocode, to the maximum extent possible. While it may not be necessary to represent equations and mathematical expressions in pseudocode, pseudocode is indispensable in illustrating the flow of algorithm execution including looping, branching, and subroutines.
  - Having the algorithm developer implement the protocol generally accelerates the turnaround time for the transition from algorithm design to simulation analysis. This approach also eliminates the loss of protocol details that can occur in first solution above. The algorithm developer can perform the protocol implementation if there is a simple and efficient application programming interface (API) for the developer to use. If the API supplies an environment that handles all the protocol-independent details of a full protocol implementation, and if the environment has a clear framework for implementing a protocol, then the algorithm developer can be responsible for the protocol implementation.

  Our experience with the STIP1-OPNET implementation showed that the first solution above was difficult to achieve, and that the series of SNAPS, NAPI, and NAPI+ APIs were immensely helpful in developing robust simulations.
- **Organize Software into Separate Protocol and Protocol-Independent Sections.** Regardless of how protocol development is accomplished, it is important to organize software subsections into protocol-independent and protocol-dependent modules when developing a number of different protocols for simulation.
- **Establish Software Configuration Control.** It is crucial to establish software configuration control when making changes to protocol implementations, even for small groups of individuals doing basic research.
- **Use Animation for Network Analysis.** Interactive animation of the history of network events is immensely helpful in understanding protocol behavior. The introduction of Netviz proved be an important development.

9.2 ACCOMPLISHMENTS

SRI developed theoretically sound, fast-responding, efficient, and robust network algorithms for multimedia networks. These new algorithms provide contingency plans for immediate reaction in case of sudden changes in hostile environments; effectively utilize the resources of multiple transmission media; and combine routing and flow control to satisfy the diverse requirements of C^3I traffic.

A modular protocol framework (NAPI+) was designed to facilitate the comparison of protocols composed of different sets of algorithms. Upon this framework we developed a series of increasingly sophisticated protocols (STIP1-3) to evaluate and demonstrate the algorithms developed. These protocols proved to perform significantly better than a simple baseline protocol and the link-state based OSPF protocol, in dynamic networks.

To help focus the analysis of the new protocols, SRI developed an environment profile that described both benign and stressful ECM network conditions and presented a methodology for quantifying network performance. We developed and evaluated ECM strategies to intelligently attack network operation, using knowledge of the network protocols; and fed back the results into the STIP protocol development cycle.

SRI developed an integrated set of network simulation and emulation tools to support the phased development and evaluation of the protocols. These tools include a suite of graphically based analysis tools (including Netviz), used for analyzing simulation results. Additionally, we demonstrated that the STIP protocols could be implemented on a multiprocessor packet switch, by developing an emulator over the Mach operating system to take advantage of parallel computation facilities.

9.3 BEYOND EDMUNDS

In the following subsections, we describe some items for future EDMUNDS-related research. In some cases, these items will be directly based on extensions to the technologies developed in EDMUNDS.

9.3.1 New Algorithm and Protocol Research

STIP3 is appropriate for a variety networks, including applications using point-to-point and microwave links.

Future research for STI protocols might include

- Protocol applications for networks other than point to point, such as broadcast, multicast, RF, and multiple access
- The use of global topology information, especially for faster links, to
  - Route on disjoint paths
  - Avoid being limited to a single DAG
  - Avoid loops
- ATM compatibility
- Bandwidth reservations and admission control
- Alternate path reservations for reliable service
- Erasure coding for the recovery of lost packets

- Dynamic adjustment of protocol parameters
- The incorporation of transmission scheduling for radio networks
- The use of neural networks for faster computation.

9.3.2 Protocol Development Environment Extensions

The following are some suggestions for extending the current NAPI+ environment:

- **New Link and Network Types**. The current NAPI+ development environment assumes a multihop multimedia network with point-to-point links. New link types such as broadcast, multicast, radio, and multiple access could be implemented.
- **Sophisticated Jamming (Link Dynamics)**. The current NAPI+ environment provides the capability to simulate jamming by specifying link dynamics. Link dynamics are specified by scheduling the up and down times for a link. The link dynamics could be extended to handle more sophisticated covert and overt jamming scenarios including
  - Models that allow the jammer to jam at the bit level
  - Built-in jamming models that free the user from using unwieldy script files
- **Spoofing**. The jamming model in NAPI+ could be extended to allow the adversary to spoof the network by injecting replicated, modified, or otherwise false packets at any location in the network. This extension should also include the capability for the adversary to listen to network traffic, a requirement needed to explore algorithms that deal with the Level III and Level IV adversaries as described in the Network Environment Profile.*
- **Multiple Protocol Layers**. The current environment allows the protocol developer to simulate only one protocol. Extensions to NAPI+ could be made, to allow for a layered protocol stack. These extensions would permit the simulation analysis of STI protocols at separate layers, and of STI protocols' behavior when they are used with other well-known protocols such as TCP and IP.
- **Multiple Node Types**. The current environment permits a network of homogeneous nodes only. NAPI+ could be extended to permit the use of multiple node types such as gateways, bridges, repeaters, and nonrouting nodes. In other words, not all nodes would be required to operate the same protocol stacks.
- **Multiple Networks**. NAPI+ currently allows the simulation of a single autonomous system. This restriction could be lifted by the implementation of features that support multiple networks.

9.3.3 Protocol Evaluation Environment Extensions

A number of extensions to the evaluation environment are possible.

---

*Lee, D.S., and D.A. Beyer. 1990. *SURAN Network Environment Profile: Benign and Adversarial Conditions, and Benchmark Scenarios*, SRNTN No. 63, SRI International, Menlo Park, California (January).

- **Interactive Graphical Simulation Control**. Further work is needed to develop an interactive graphical system for controlling and analyzing simulations. It is important to develop a system where the experimenter could control the link dynamics as the simulation executes, and immediately see the results in Netviz, as described below.
  - **Front-End Simulation Controller.** A front-end controller would be part of a system that would allow the experimenter to interact directly with the simulation as it executes, as opposed to feeding a script file to the simulation. The controller would display the network topology on the workstation screen and allow the user to bring links up and down by clicking on them with the mouse. In addition to being useful for demonstrations, the controller would be immensely useful in the immediate exploration of questions that arise during the course of a simulation execution.
  - **Integrating Simulations with Netviz**. To complete this system, some work would have to be done to the simulation environment to provide the output that is required for driving the Netviz animations.
- **Multiple Parallel Links**. The EDMUNDS analysis tools should be extended to handle more diverse network scenarios. For example, some of the analysis tools do not support multiple parallel links between two nodes, regardless of whether or not the parallel links are of the same media.
- **Protocol-Independent Analysis Tools**. Some of the analysis tools, such as the nv filters used for driving Netviz animations, are not very flexible. These filters should be modified so that they do not require the filters to be changed for each new parameter that the experimenter wishes to animate.

10 ACRONYMS AND ABBREVIATIONS

ack	Acknowledgement; Acknowledge
adr	Address
alg	Algorithm
AMIDS	Advanced Multimedia Information Distribution System
ARPA	Advanced Research Project Agency
BF	Bellman Ford [routing] (a distributed routing algorithm like that used in SURAN)
bps	Bits per second
Bps	Bytes per second
CELP	Codebook Excited Linear Predictive [speech coding]
cntrl	Control
CPU	Central Processing Unit
DAG	Directed Acyclic Graph
DARPA	Defense Advanced Research Projects Agency (now ARPA)
DARTNET	DARPA Research Testbed Network
dev	Device
dst	Destination (usually the final destination)
EBS	EDMUNDS Benchmark Scenario
ECP	Extended area Command Post
EDMUNDS	Evaluation and Development of Multimedia Networks in Dynamic Stress
ER	Extended-area Repeater
ES	Extended-area Sensor
ETE	End to end
FCFS	First Come, First Served
info	Information
IP	Internet Protocol
ISDN	Integrated Services Digital Network
LAV	Link Attribute Vector (used in emulators)
LCP	Localized-Area Command Post
LF	Localized-Area Fighting Unit

LM	LF Mobile
LPC10	[10-order] Linear Predictive Coder (for speech)
LR	Localized-Area Repeater
LSA	Link State Advertisement
MEDAR	Minimum Expected Delay Alternate Routing algorithm
Mbps	Mega (Million) bits per second
NAPI	Network Algorithm Programmer's Interface
NAV	Node Attribute Vector (used in emulators)
nbr	Neighbor
net	Network
NSA	National Security Agency
OO	Object Oriented
OS	Operating System (e.g., UNIX, SunOS, and Mach)
OSPF	Open Shortest Path First (an Internet routing protocol)
PFP	Parameter File Parser (software library for parsing parameters in .ef files)
pkt	Packet
PR	Packet Radio
PSRT	Packet Speech Research Terminal
q	queue
QMgmt	Queue mar ement
rcv	Receive
RF	Radio Frequency
RL	Rome Laboratory
rte	Route
RTNP	Robust Transfiguring Network Protocols
SAINT	Shared Adaptive Internetworking Technology
seq	Sequence
SNAPS	Simple Network Algorithm Prototyping Emulator
SPF	Shortest Path First routing (decentralized algorithm used in Internet)
src	Source
SSS	SURAN Standard Scenario

STI	Secure Tactical Internet
STIPn	STI Protocol n (1, 2, or 3)
SURAN	Survivable Adaptive Networks program funded by ARPA
TCP	Transmission Control Protocol
TDMA	Time Division Multiple Access
TG	Traffic Generator
TOS	Type of Service (individualized handling of different traffic types)
WCP	Wide-area Command Post
xmt	Transmit

BIBLIOGRAPHY

BDM Corporation. 1988. *Data Link and Network Layer Protocols for Army Tactical Command and Control.* Technical Report BDM/TAC-88-152-TR, BDM Corporation, Tacoma, Washington (May).

Bertsekas, D., and R. Gallager. 1987. *Data Networks.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey.

Beyer, D.A. 1991. *SURAN Standard Scenario (SSS) Generator User's Guide*, SURAN Program Technical Note 66, SRI International, Menlo Park, California (February).

Beyer, D.A., J.M. Hight, R.G. Ogier, and D.S. Lee. 1993. *Secure Tactical Internet Protocol 1 (STIP1)*, ITAD-8558-TR-93-31, SRI International, Menlo Park, California (February).

Brennan, W. 1986. *Network Performance of Dynamic UHF Radio Networks*, Technical Report TR-86-1, U.S. Army Communications-Electronics Command, Fort Monmouth, New Jersey (February).

Brennan, W. 1987. "Dynamics of Mobile Networks," viewgraph handout, SURAN Working Group Meeting, Menlo Park, California (January).

Bucholz, D. 1987. *User Interface Requirement for Force Level Control System.* Technical Report Memorandum, U.S. Army Combined Arms Combat Development Activity (CADA), Fort Leavenworth, Kansas (June).

CECOM. 1988. *Test and Evaluation Master Plan H (TEMP-II) for EPLRS,* Technical Report prepared by PLRS/TIDS Program Manager (July).

Dacunto, COL. 1987. *Army Command and Control Master Plan - Vol. 1, Concept and Management,* Technical Report, U.S. Army Combined Arms Combat Development Activity (CADA), Fort Leavenworth, Kansas (August).

Fultz, G.L., and Kleinrock, L. 1971. "Adaptive routing techniques for store-and-forward computer-communication networks," in *Proc. ICC*, pp. 39.1–8.

Gallager, R.G. 1977. "A minimum delay routing algorithm using distributed computation," *IEEE Trans. on Communications*, COM-25(1):73–85 (January).

Garretson, J. 1988. *Independent Evaluation Report (Interim) of the Mobile Subscriber Equipment (MSE) Follow-on Operational Test and Evaluation,* Technical Report, U.S. Army Operational Test and Evaluation Agency, Falls Church, Virginia (August-October).

Hajek, B., and R.G. Ogier. 1984. "Optimal dynamic routing in communication networks with continuous traffic," *Networks*, 14:475–487.

Hann, D., and B. Caffall. 1988. *Detailed Test Plan (Section 1 and 2), Development Test HB (PQT-G) of EPLRS,* Technical Report USAEPG-TP-1302, Vol. 1, U.S. Army Electronic Proving Ground, Fort Huachuca, Arizona (April).

Hann, D., and B. Caffall. 1988. *Detailed Test Plan (Section 3), Development Test HB (PQT-G) of EPLRS,* Technical Report USAEPG-T?-1302, Vol. I, U.S. Army Electronic Proving Ground, Fort Huachuca, Arizona (April).

Harrick, G. 1990. *ADI Communications System Design Study*, Final Technical Report RADC-TR-90-110, Vol. 1, Harris Corporation for Rome Air Development Center (June).

Hughes Aircraft Co. 1988. *Enhanced Position Location Reporting System (EPLRS) Technical Description*, Technical Report CDRL-06AV-001A, Hughes Ground Systems Group, Fullerton, California (May).

Hughes Aircraft Co. 1989. Discussions with Steve Witt (May).

International Organization for Standardization. 1988. *Security Architecture*, ISO 7498/2.

ITT. 1985. *SINCGARS Packet Radio Multiple User Performance Analysis*, Technical Report, ITT Technical Note to M. Pursley and A. Shohara (December).

Jones, Captain C. 1990. "Multimedia Communications in the Air Defense Environment," presented at MILCOM '90 (October).

Kota, S., and J. Garcia-Luna-Aceves. 1990. "Survivable Planar Internetwork (SPIN) for Multimedia Communications," presented at MILCOM '90 (October).

Lee, D.S., and D.A. Beyer. 1990. *SURAN Network Environment Profile: Benign and Adversarial Conditions, and Benchmark Scenarios*, SRNTN No. 63, SRI International, Menlo Park, California (January).

Lee, D.S., D.A. Beyer, and R.G. Ogier. 1992a. *EDMUNDS Network Environment Profile: Benign and Adversarial Conditions, and Benchmark Scenarios (Part I)*, ITAD-8558-TR-91-23, SRI International, Menlo Park, California (May).

Lewis, M. 1987. *SINCGARS Packet Applique*, Technical Report, SRI International, Menlo Park, California.

Luenberger, D.G. 1969. *Optimization by Vector Space Methods*, John Wiley and Sons, Inc., New York.

Mathis, J. 1986. *SINCGARS Packet Switch Overlay*. Technical Report, SRI International, Menlo Park, California (July).

McDermott, E., and G. Lauer, et al. 1988."SURAN and Tactical Network Security (U)," presented at Tactical Communications Conference (May).

McKenney, P., and R.G. Ogier. 1989. *Measures of Effectiveness For Quantitatively Analyzing Dynamic Network Performance*, Technical Report, SRI International, Menlo Park, California.

Nelson, R. 1987. "SDNS Services and Architecture," *Proc. 10th National Computer Security Conference*, GTE Government Systems Corporation, Baltimore, Maryland (September).

Ogier, R.G. 1988. "Minimum-delay routing in continuous-time dynamic networks with piecewise-constant capacities," *Networks*, 18:303–318.

Ogier, R.G., and D.S. Lee. 1993. *Secure Tactical Internet Protocol 2 (STIP2)*, ITAD-8558-TR-93-320, SRI International, Menlo Park, California (September).

Ogier, R.G., and V. Rutenberg. 1992a. "Minimum Expected Delay Alternate Routing (MEDAR)," INFOCOM '92.

Ogier, R.G., and V. Rutenberg. 1992b. "Robust Routing for Minimum Worst Case Delay in Unreliable Networks," INFOCOM '92.

Pursley, M.B. 1987. *Survey of Research on Network ECM/ECCM.* Technical Report prepared for JPL, University of Illinois (September).

Rudin, H. 1976. "On routing and 'delta routing': A taxonomy and performance comparison of techniques for packet-switched networks," *IEEE Trans. on Communications*, COM-24(1):43–59 (January).

Rutenberg, V., and R.G. Ogier. 1993. "How to Extract Maximum Information from Event-Driven Topology Updates," INFOCOM '93.

Schwartz, M. 1986. *Telecommunication Networks: Protocols, Modeling and Analysis*, Addison-Wesley Publishing Co., Reading, Massachusetts.

SRI. 1989. Discussions with Bill Ficklin (May).

U.S. Army. 1977. *Communications Jamming: An Element of Combat Power*, Technical Report TC30-12-1, Headquarters Department of the Army (October).

U.S. Army. 1984. *Armored and Machinized Division and Brigade Operations*, FC 71-100, U.S. Army Command and General Staff College, Fort Leavenworth, Kansas (May).

U.S. Army. 1987. *ADDS EPLRS Test Design Plan (TDP)*, Technical Report, U.S. Army Materiel Systems Analysis Activity, Aberdeen Proving Ground, Maryland (May).

U.S. Army. 1988. "Corps/Division Operations," Military Review PB-100-88-7, U.S. Army Command and General Staff College, Fort Leavenworth, Kansas (July).

U.S. Army. 1988. "Signal Support in the AirLand Battle," FM-24-1, HQ Department of the Army, Washington, D.C. (December).

U.S. Army. 1988. *System Description for the Army Tactical Command and Control System (ATCCS)*, ACCS-AI-100-001, Army Tactical Command and Control Program (March).

U.S. Army. 1989. "An Analysis of Battlefield Communications," prepared for the U.S. House of Representatives Committee on Armed Forces, HQ Department of the Army, Washington, D.C. (January).

U.S. Army. 1989. "Army Command and Control Communications Intelligence Electronic Warfare Master Plan," Revision 1 (November).

U.S. Army. 1989. "Battlefield Information System 2015," U.S. Army Communications-Electronics Command, Fort Monmouth, New Jersey (September).

U.S. Army. 1990. *Army Command and Control Master Plan (AC2MP)*, Combined Arms Combat Developments Activity (CACDA), Fort Leavenworth, Kansas.

Williamson, J. (editor). 1989. *Jane's Military Communications Systems*, 10th Edition. Jane's Information Group Limited, Alexandria, Virginia.

Yavuz, D. 1990. "Meteor Burst Communications," *IEEE Communications Magazine* (September).

Appendix A

NETWORK ENVIRONMENT PROFILE

NETWORK ENVIRONMENT PROFILE

The objective of the Evaluation and Development of Multimedia Networks in Dynamic Stress (EDMUNDS) program is to analyze algorithms that provide reliable, efficient, fair, and robust service in multimedia networks despite mobility, changing traffic loads, and sophisticated attempts to penetrate or disrupt the network. The algorithms and resulting protocols developed must permit continued service despite these conditions.

After development, the strength of the algorithms and protocols will be determined through performance and vulnerability analysis efforts. Performance analysis includes determination of throughput and delay of end-to-end (ETE) traffic under conditions of varying degrees of stress (e.g., traffic load and number of mobile nodes). Vulnerability analysis will focus on determining the effects of electronic countermeasures (ECMs) and network monitoring on the disruption of network-level performance.

In order to guide the algorithm and protocol development and analysis efforts, it is necessary to identify the network conditions of concern to EDMUNDS. In addition, these guidelines list other conditions that are not being addressed. The resulting environment profile is divided into two parts, an unclassified part (Part 1) and a classified part (Part II*). Part I describes a framework of network conditions, and benchmark scenarios for the EDMUNDS development and analysis effort. A methodology for determining the performance of the protocols and their vulnerability (i.e., the effectiveness of adversary attack strategies on protocols) has been developed and is included in Part I. In Part II, a description of the adversary's electronic countermeasure capabilities provides a guide for conducting vulnerability analysis experiments. Though these network environment guidelines are being used to provide a framework for the EDMUNDS algorithm and protocol development and analysis efforts, performance and vulnerability analyses are also expected to probe beyond these guidelines in an attempt to determine the performance boundaries of the algorithms and protocols.

This unclassified document covers Part I of the environment profile and is divided into two main sections. The first section describes benign and unsophisticated adversarial network conditions including benchmark scenarios; and the second section describes a methodology for estimating protocol performance and the effectiveness of adversary attacks on network performance. Part II of the environmental profile describes conditions caused by a sophisticated adversary.

A.1 BENIGN AND UNSOPHISTICATED ADVERSARIAL CONDITIONS

In this section, we describe conditions for the benign EDMUNDS network environment and for the EDMUNDS network environment influenced by an unsophisticated adversary. This profile is used to guide the development and analysis of the EDMUNDS algorithms and protocols. It is also used as a framework to build standard or "benchmark" network scenarios for evaluating and comparing the protocols being developed under EDMUNDS.

---

*Lee, D.S., D.A. Beyer, and R.G. Ogier. 1992. *EDMUNDS Network Environment Profile: Benign and Adversarial Conditions, and Benchmark Scenarios (Part I)*, ITAD-8558-TR-91-23, SRI International, Menlo Park, California (May).

The benign and unsophisticated adversary EDMUNDS network profile is defined for the following network environment parameters:

- Nodal Processing—processing power
- Transmission Media—type and characteristics
- Topology-network size and dynamics
- Traffic distribution and pattern
- Jamming-pattern.

Subsection A.1.1 describes the boundary conditions for these network environment parameters that should be considered during the algorithm development. Lee and Beyer* describe a similar environment profile for the Survivable Adaptive Networks (SURAN) programs, the referenced report can be used to further detail the network conditions within the boundaries identified here; tools and algorithms for creating scenarios and correcting for partitioned networks are also described. The benchmark network scenarios used for evaluation and comparing the protocol packages provide specific conditions for the network environment, within the boundaries defined, and are described in Subsection A.1.2. It should be noted that these benchmarks are not intended to replace more complete sensitivity analysis under conditions appropriate for the protocol being evaluated.

A.1.1 General Network Environment

This section describes the general conditions for the EDMUNDS network environment profile.

A.1.1.1 Nodal Processing

We assume that advanced processing capabilities of between 100 and 200 MIPs are present at each node, to reflect the estimated processing power available in the future.

A.1.1.2 Transmission Media

The EDMUNDS network consists of diverse transmission links—its network nodes are interconnected by various communications media. Communications include both point-to-multipoint (broadcast) and point-to-point links. Communication channels with throughput between 1 kbps and 1 Mbps are emphasized.

A significant portion of the network is composed of RF channels including VHF, UHF, HF sky wave, satellite, meteor burst, and troposphere. Radio communications include both single- and multichannel, and both directional and omnidirectional antenna systems. Finally, land lines such as fiber optics are used to interconnect various EDMUNDS network elements.

A.1.1.3 Topology

The following describes benchmark options for the EDMUNDS network topology.

Network Size—Emphasis is given to networks of moderate size (50 to 1000 nodes).

---

*Lee, D.S., and D.A. Beyer. 1990. *SURAN Network Environmental Profile: Benign and Adversarial Conditions, and Benchmark Scenarios*, (SRNTN No. 63), SRI International, Menlo Park, California (January).

Dynamics—Focus is given to very high dynamics. The effects of mobility will be considered, but will not be the primary cause of dynamics, due to their relatively long link lifetimes of 10 s or more. Dynamics will focus on jamming where periods can be as small as a packet size (e.g., 1 ms for a 100-bit packet over a 100-kbps channel).

A.1.1.4 Traffic

Traffic types to consider include voice, file transfers, images, database updates, data messages, and sensor data. In current deployments, voice traffic consumes approximately 75% of the communication resources. The EDMUNDS model will reflect increased requirements for nonvoice traffic as it becomes prevalent in the battlefield. For the benchmark scenarios, nonvoice data constitutes 67% of the traffic load—see Subsection A.1.2.5.

The traffic will have diverse throughput, delay, and reliability requirements and varying degrees of burstiness. Typical data rates for voice will be 1 to 4.8 kbps, characterized by bursty traffic with random, exponentially distributed bursts. Data traffic for file transfers and images consists of 5- to 30-kbyte messages with random, exponentially distributed intermessage times; database updates and data messages consist of a 1-kbyte message with random, exponentially distributed intermessage times; and sensor data consists of 100-byte messages with a constant rate.

A.1.1.5 Jamming

The jamming capabilities of the unsophisticated adversary are very limited, since no knowledge of the protocols or monitoring capabilities are assumed of the adversary. The adversary basically knows only the location of the nodes, nodal hardware capabilities, and general network mission. See Part II* of this report for a description of the sophisticated adversary.

The adversary uses both narrow-beam (nodal) and wide-beam (60°) jammers. The jamming patterns are generally restricted to random nodal and area-sweep jamming where the jamming signals are pulsed in time or transmitted continuously. As described above, jamming periods can be as small as the length of a transmitted packet.

A.1.2 Benchmark Scenarios

Based upon the general conditions described above for the EDMUNDS network environment, a set of benchmark scenarios has been defined, including communications support areas, network size, nodal placement and identification, link assignment, traffic stream assignment, and jamming. It is intended that the appropriate subset of these benchmark scenarios provide the minimum test conditions required for each protocol developed in EDMUNDS. Although not all of the benchmark scenarios are appropriate for testing each protocol, some subset should be. In addition, it is not intended that these benchmarks replace more complete sensitivity analysis under conditions appropriate for the protocol being evaluated. The benchmark scenarios will also be used to facilitate comparison between different protocols and to provide a baseline for comparison with the results of the EDMUNDS vulnerability analysis task.

---

*Lee, D.S., D.A. Beyer, and R.G. Ogier. 1992. *EDMUNDS Network Environmental Profile: Benign and Adversarial Conditions, and Benchmark Scenarios (Part II)*, ITAD-8558-TR-91-23, SRI International, Menlo Park, California (July), SECRET.

The set of benchmark network options has been developed in an attempt to form a best fit of the following conflicting needs. The options should

- Be few in number
- Be specified in enough detail to permit comparisons of the relative performance advantages and disadvantages of protocols, independent of particular experiment environments
- Model scenarios that are likely to be found in realistic deployments, so that the resulting measurements can be used to indicate real-life performance
- Provide maximal coverage of the multidimensional space defined by all possible, realistic scenarios to permit comparisons under different extremes.

In addition, the benchmarks have been designed in an attempt to satisfy the following practical considerations:

- The scenarios should be simple to describe and easy to automate.
- Some subset must define conditions that can be simulated today, in a "reasonable" amount of time (about 10 hours at most for overnight experiments), using the network emulator being developed under EDMUNDS.
- Because it is not known today exactly how the network will be used in the future, the benchmarks should define traffic and dynamic link conditions that cause the network to operate around its limits. To some extent, trial and error must be used to determine such conditions.

Keeping in mind the above constraints, the benchmark scenarios defined below are derived, in part, from possible tactical deployments similar to those used for systems such as Single Channel Ground and Airborne Radio Systems (SINCGARS), Mobile Subscriber Equipment (MSE), and Improved High-Frequency Radio (IHFR) and are based upon the Army echelon composition model*. Although the scenarios are Army oriented, they are general enough to be extensible to other military forces. Six benchmark scenarios are identified, built upon the communications support areas and based on the number of nodes in the network.

A.1.2.1 Communications Support Areas

It is expected that command and control communications of the future will be a graphically dispersed system, requiring different types of communications support. We define three support areas—localized, extended, and wide. These three support areas are used to build the multimedia communications network for the various benchmark scenarios. Typically, localized areas are grouped together to form an extended area, and extended areas are then grouped together to form a wide area. Table A-1 describes each of these areas and their associated characteristics, including area coverage and communications media along with link range and capacity.

*U.S. Army. 1984. *Armored and Mechanized Division and Brigade Operations*, FC 71-100, U.S. Army Command and General Staff College, Fort Leavenworth, Kansas (May).

Table A-1. Multimedia Communications Support

SUPPORT AREA	AREA COVERAGE (W × D IN MILES)	MEDIA	LINK RANGE (MILES)	LINK CAPACITY
Localized (L)	30 × 100	VHF UHF	0-60 0-40	16 kbps 400 kbps
Extended (E)	150 × 100	Fiber Optics Troposcatter Satellite	0-100s 32-240 0-10,000	≥ 100 Mbps 2 Mbps 2,400 Mbps
Wide (W)	1000 × 1000	Meteor Burst HF Sky Wave Satellite	400-900 100s-2,500 0-10,000	100-kpbs Burst Rate 5% Duty Cycle 2,400 bps 2,400 bps

The **localized** area (L) supports fighting units and their command posts; these nodes are generally close in proximity and are typically mobile. Communications media used within the localized area are typically broadcast, line-of-sight systems such as UHF and VHF radios. A localized area is defined to be 30 miles wide by 100 miles deep. Although larger in area than brigade and below, the localized area provides similar functionality.

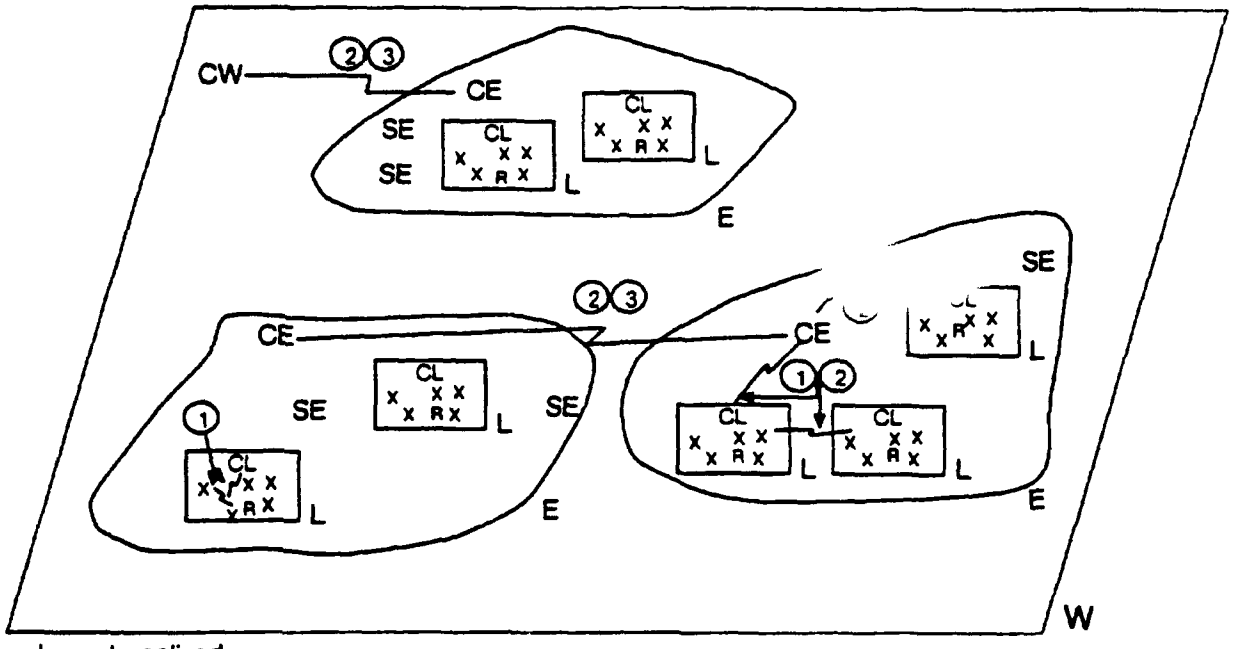
The **extended** area (E) is used to provide connectivity across echelons and extends communications to all levels of fighting forces; the nodes are dispersed over an area of 150 miles by 100 miles, typifying the corps and below. Communications media used within an extended area include fiber optics, troposcatter, and satellite. These media are used to link localized areas (within an extended area) and for communications amongst localized and extended area command posts. Communications between these nodes is also supplemented by transmission media supporting localized areas.

Finally, the **wide** area (W) provides backbone communications, spanning the theatre and beyond. Communication is basically point-to-point, covering an area as wide as 1000 miles by 1000 miles. HF sky wave, meteor burst, and satellite are used at this level to connect command posts of different extended and wide areas. Communication between these nodes is also supplemented by transmission media supporting extended areas.

In addition to the fighting units (FUs) and command posts (CPs) at each of the communications support areas, sensor nodes (S) and relay nodes (R) are also supported by the transmission media described above, as appropriate. Figure A-1 shows how the communications support areas and various network nodes are configured to form a basic multimedia communications tactical network.

A.1.2.2 Network Size

We define six benchmark scenarios based upon network size and built from the three communications support area described above. The following network sizes reflect information extracted from a number of military publications, especially a report by J. Garretson 1988.* Determination of network size is modular in approach, and can be easily adjusted to satisfy other requirements.



L = Localized
E = Extended
W = Wide Area
X = Fighting Unit
R = Relay Node
C = Command Post
S = Sensor Node

Media Group
1. UHF, VHF
2. Fiber, Tropo, Satellite
3. Meteor Burst, HF Sky Wave

Figure A-1. Multimedia Communications Tactical Network

As mentioned above, the EDMUND S program will emphasize networks of moderate size (50 to 1000 nodes). Table A-2 presents the six network sizes for the benchmark scenarios—one covering a localized area, two covering an extended area, and three covering a wide area. Those numbers in parentheses are not part of the network sizes considered, but are shown to demonstrate how the network sizes are derived.

Table A-2. Network Size

COMMUNICATIONS SUPPORT	NUMBER OF NODES		
Localized (L) (= 75% FUs + 25% Rs)	(1)	(20)	40
Extended (E) (= 5Ls + 1 CPE + 3 SEs + 2Rs)	(11)	(106)	206
Wide (W) (= 5 Es + 1 CPW)	56	531	1,031

*Garretson, J. 1988. *Independent Evaluation Report (Interim) of the Mobile Subscriber Equipment (MSE) Follow-on Operational Test and Evaluation*, Technical Report, U.S. Army Operational Test and Evaluation Agency, Falls Church, Virginia (August-October).

The network sizes are determined by identifying the number of nodes in a localized area and using this as a building block to create larger networks. Basically, in addition to a few specialized nodes (i.e., command posts, sensor nodes, and relays), a localized area consists of 75% fighting units and 25% relays; an extended area consists of five localized areas; and a wide area consists of five extended areas.

As an example, the following describes the construction of a 1031-node network. This network covers a wide area and consists of five extended-area plus one wide-area command-post node (CPW); each of the extended areas consists of five localized areas, one extended-area command post node (CPE), three sensor nodes (SE), and two relay nodes; each localized area consists of 39 nodes, of which 75% are fighting units and the remaining are relay nodes and one localized-area command-post node (CPL).

Two network scenarios have been defined in the 40- to 60-node range. The 40-node network enables testing of a network with broadcast media only, and the 56-node network allows for testing of all media using a small-sized network.

A.1.2.3 Nodal Placement and Identification

The general approach for placing the network nodes is to first lay down the wide area, randomly lay down the extended areas, and then lay down the localized areas. The nodes (fighting units, command posts, sensors, and relays) are then placed into their respective support area with the command post nodes at the far edge of their respective support area.

Specifically,

1. Identify one edge of the wide area (W) as the near edge.
2. Randomly lay extended areas (Es) into the wide area (W) such that the 150-mile edge of E is parallel to the near edge of W and the Es do not overlap. Let the edge of E closest to the near edge of W be the forward line of troops (FLOT).
3. For each E, divide E into 5 rectangular areas (of equal size) running the depth of E. Each of these rectangular areas is a localized area (L).
4. For each L, place the fighting units and relay nodes so that they are uniformly distributed along the width (w) and exponentially distributed along the depth (d). The exponential distribution is defined as follows: $d = e^{4.6r}$ where $0 \leq r \leq 1$ is uniformly random and $1 \leq d \leq 100$ represents linear values along the depth of L with 1 at the edge closest to the FLOT and 100 at the edge furthest from the FLOT.
5. For each L, select from the fighting units a command post node that is furthest from the FLOT.
6. For each E, randomly place a command post node along the edge furthest from the FLOT and randomly place 3 sensor nodes and 2 relay nodes into E.
7. For W, randomly place a command post node along the edge opposite its near edge.

A.1.2.4 Link Assignment and Connectivity

Table A-3 identifies the possible links between the benchmark network nodes and the potential transmission media supporting these links. We will determine link connectivity based upon a specified percentage of nodes that are within the communications range. Connectivity will depend upon the media characteristics and distance of the nodes. The following defines the network links for the benchmark scenarios:

- VHF and UHF links will be connected depending upon attenuation (e.g., path loss plus multipath fading).
- Of the potential troposcatter, meteor burst, and HF sky wave links within distance range (see Table A-1), 50% each will be connected.
- Of the potential fiber-optic links within distance range (see Table A-1), 33% will be connected.
- Of the potential satellite links, 100% will be connected.

A.1.2.5 Traffic Stream Assignment

Traffic stream assignment defines a source/destination pair along with the traffic type transmitted over the link. Because we want to specify traffic conditions that adequately load the network, we plan to run experiments using the benchmark scenarios to more accurately specify appropriate traffic distributions.

Table A-4 defines the possible traffic types over these source/destination pairs along with their traffic distribution. The traffic distribution quantities are based upon the assumption that the traffic profile for each source/destination pair (excluding links to sensor nodes) is identical and characterized by 1-kbps traffic throughput consisting of equal proportions of the three principal traffic types (e.g., not including sensor traffic) for each link. One hundred bytes/1.0 s means that the traffic consists of 100-byte messages transmitted periodically every 1 second. One kbyte/ [24 s (exp)] means that the traffic consists of 1-kbyte messages transmitted with an exponentially distributed interpacket time with a mean of 24 seconds. The traffic types are named as the real-world that they represent; however, a number of approximations are used. For example, end-to-end transport acknowledgments which would be correlated to the traffic in the real-world, are approximated by bidirectional random (uncorrelated) traffic streams.

Table A-3. Potential Links and Media

LINK	MEDIA
Fighting Unit ⟷ Fighting Unit Fighting Unit ⟷ Localized CP	VHF, UHF
Localized CP ⟷ Localized CP Localized CP ⟷ Extended CP	VHF, UHF, Fiber, Tropo, Satellite
Extended CP ⟷ Extended CP Extended CP ⟷ Wide CP	Fiber, Tropo, Satellite, Meteor, HF Sky Wave
Extended CP ⟷ Sensor	Fiber, Tropo, Satellite

Table A-4. Benchmark Options for Traffic Type and Distribution

TRAFFIC TYPE	TRAFFIC DISTRIBUTION
Database Update or Data Message	1 kybte/[24 s (exp)]
File Transfer or Images	20 kbytes/[8 m (exp)]
Voice	24 kb/[1.2 m (exp)]
Sensor	100 bytes/l.O s

Table A-5 specifies which nodes transmit data between one another. Not all fighting units transmit data to all other fighting units nor to command posts in any localized area; fighting units communicate only with other nodes within their localized area. Similarly, localized (area) command posts communicate only with fighting units within their localized areas and with localized command posts and extended (area) command posts within their extended area; extended command posts communicate with localized command posts within their extended area and with all extended and wide (area) command posts; and extended command posts receive sensor information only from sensors within their extended area.

Table A-5. Traffic Streams

SOURCE ◄——► DESTINATION		
Fighting Unit	◄——►	Fighting Unit within Localized Area
Fighting Unit	◄——►	Localized CP within Localized Area
Localized CP	◄——►	Localized CP within Extended Area
Localized CP	◄——►	Extended CP within Extended Area
Extended CP	◄——►	Extended CP
Extended CP	◄——►	Wide CP
Extended Sensor	——►	Extended CP within Extended Area

A.1.2.6 Jamming

As mentioned previously, EDMUNDS is focusing on high dynamics where jamming rates can be extremely fast. For the benchmark scenarios, 25% of the links of each RF media (i.e., VHF, UHF, and HF sky wave) will be randomly selected and jammed for a period of t where $t \in$ (10 ms, 100 *ms*, 1 *s*, 10 *s*). After each period t, the RF links are again randomly selected.

A.1.2.7 Benchmark State Transition

The network and protocol performance will be evaluated at two bench[illegible] states. The scenarios described above will be used as the initial test state. As a specific [illegible] t, a state transition, defined by the removal of all fiber-optic and satellite links and rela[illegible]odes, will occur.

Removal of all relay nodes realistically simulates a post-attack degraded state while maintaining the same end-to-end traffic profile. The network performance will be evaluated during both the initial and degraded states, as well as during the transition.

A.2 METHODOLOGY FOR ESTIMATING PROTOCOL PERFORMANCE AND VULNERABILITY IN BENCHMARKS

A methodology is needed to quantize the results of the protocol tests and experiments in order to evaluate the performance of the protocols and their vulnerability against various red team ECM attacks. Performance analysis will characterize the network in the absence of malicious interference. In contrast, vulnerability analysis will focus on the effectiveness of the adversary attacks on the network. Effectiveness on the adversary attack must take into account both the cost and sophistication of the attack and its effect on network performance. Attacks that result in the greatest degradation of service for the lowest cost and lowest level of sophistication are exposing the greatest vulnerabilities in the protocol.

In this section, we first describe several performance measures, which are then used to define a performance measurement (P) that can be used to determine the overall performance of the protocol package. We also present a methodology, based upon P and the cost approximation C as described in Part II of *EDMUNDS Network Environment Profile: Benign and Adversarial Conditions, and Benchmark Scenarios* (see Subsection A.1.1.5), for evaluating the effectiveness on ECM attacks to the EDMUNDS protocols.

A.3 PERFORMANCE MEASURES

Network performance can be defined by a number of different criteria, all of which should be realistic descriptors of the protocols' performance. Below, we define four performance criteria (throughput, delay, reliability, and fairness) that can be used to compare various characteristics of the algorithms and protocol packages developed. These criteria are based in part upon those developed by McKenney and Ogier.* The combined performance measurement defined in Subsection A.3.4 takes into consideration all four of the criteria; this measurement can be used to determine the overall performance of the protocol package and effectiveness of various attack strategies.

A.3.1 Throughput

The throughput criterion is represented as

$$\frac{1}{T}\int_0^T \sum_s R_{s,t} \qquad \text{(Eq. 1)}$$

where $R_{s,t}$ is the rate of successfully received data bits for stream s at time t and T is the length of the experiment. The term stream refers to a sequence of packets traveling from a single source to a single destination, possibly over multiple paths through the network.

*McKenney, P., and R. Ogier. 1989. *Measures of Effectiveness for Quantitatively Analyzing Dynamic Network Performance*, Technical Report, SRI International, Menlo Park, California.

A.3.2 Delay

The delay measurement is given by

$$\frac{1}{N_a}\sum_i (X_i + Q_i) \qquad \text{(Eq. 2)}$$

where N_a is the total number of user data packets that successfully arrive at their destination, X_i is the time that packet i spent traversing links, and Q_i is the time packet i spent in queues.

A.3.2.1 Reliability

Reliability is represented as

$$\frac{N_a}{N_s} \qquad \text{(Eq. 3)}$$

where N_a is the total number of distinct user data packets that successfully arrive at their destination and N_s is the number of distinct user data packets that were sent from the source. Because some packets may stay in the network for a very long time, we will choose some large delay D and define N_a as the number of packets whose delay is less than D.

A.3.3 Fairness

Our definition of fairness is based upon a model for performance measurement that takes into account throughput, reliability, and delay. This model is described below and is followed by the definition of fairness.

Model. Different traffic types have different requirements; we characterize each traffic type (given a desired throughput) by the following four parameters:

t_1: Maximum desired delay

f_1: Maximum desired delay factor

t_2: Maximum tolerable delay

f_2: Maximum tolerable delay factor.

If a packet arrives at the destination with delay $d \le t_1$, it receives a perfect performance score (=1.0); if it arrives with delay $d \ge t_1$. it receives a zero; if it arrives with delay $t_1 \le d \le t_2$, it gets a linearly decreasing score between (f_2) and (f_1) computed as follows:

$$f_1 - (d - t_1)\frac{(f_1 - f_2)}{(t_2 - t_1)} \qquad \text{(Eq. 4)}$$

Each traffic type would be assigned particular values for t_1, t_2, f_1, and f_2 according to its requirements. Table A-6 defines possible values for the benchmark traffic types.

Table A-6. Traffic Type Characteristic

TRAFFIC TYPE	t_1	f_1	t_2	f_2
Database Update or Data Message	1 s	1.0	10 s	0.5
File Transfer	1 s	1.0	5 s	0.5
Images	2 s	1.0	10 s	0.0
Sensor	1 s	1.0	5 s	0.0
Voice	250 ms	1.0	1 s	0.5

Definition. Using the above performance measurement, we define fairness as follows:

$$\frac{1}{\sum_s a_s} \sum_s a_s \frac{(\overline{M} - M_s)^2}{(\overline{M})^2} \qquad \text{(Eq. 5)}$$

where M_s is the average of the performance of each distinct packet that stream s desired to transmit; M is the average over all M_s; and a_s represents the "importance" of each stream (we plan to initially set a_s=1 for all s).

This measurement emphasizes the most unfortunate users; i.e., if a few users get very poor service or if a few users get very good service, then the system is very unfair.

A.3.4 Measuring Protocol Performance

Although it is important to note performance of the protocols for each of the four measures above, a single, combined performance measure is needed to estimate the overall performance of the protocol package.

The following combined performance measure (P) takes into account throughput, delay, reliability, and fairness and builds upon the techniques developed for the fairness measurement. The measurement is

$$P = 1 - \frac{1}{S} \sum_s (1 - M_s)^2 \qquad \text{(Eq. 6)}$$

where M_s is the average of the performance of each distinct packet that stream s desired to transmit and S is the total number of streams. P is bounded between 0 and 1, with 1 representing a perfect score.

This measure has the following desirable properties:

- If the throughput on one stream improves, then P increases.
- If fairness improves while the average performance over all sessions stays constant, then P increases.
- If the average performance over all sessions improves while fairness stays constant, then P increases.

A.4 MEASURING PROTOCOL VULNERABIUTY

To determine the effectiveness of attack strategies on different protocol packages for a particular benchmark scenario, we could ask *What cost is required to effect an x% decrease in performance?* and then compare this cost with that required for different protocol packages. However, because it is difficult to predict the performance of a particular attack, it is better to ask the following: *Given a cost allocation of x, what percentage decrease in performance can be achieved?*

The methodology to be used for measuring the vulnerability of each protocol package is as follows:

- Determine the performance P that can be achieved under a specific attack strategy with costs C equal to 0.05, 0.1, 0.25, 0.5, and 0.75. Let P be the combined performance measure as described in Subsection A.3.4. Cost C is defined in Part II of *EDMUNDS Network Environment Profile: Benign and Adversarial Conditions, and Benchmark Scenarios* (see Subsection A.1.1.5) as the cost associated with the effectiveness of an adversary attack; initially set $W_i(t)$ (power of jammer i at time t) to 1 for all jammed nodes.
- Graph the results on a curve such as that presented in Figure A-2. Recall that performance P is between 0 and 1, with 1 representing a perfect score.

  Graph the results on a curve such as that presented in Figure A-3.
- Repeat for other benchmark scenarios.
- Repeat for other attack strategies.
- Also, compute the attack effectiveness (E) for each attack as follows:

$$E = \frac{1 - P}{C} \qquad \text{(Eq. 7)}$$

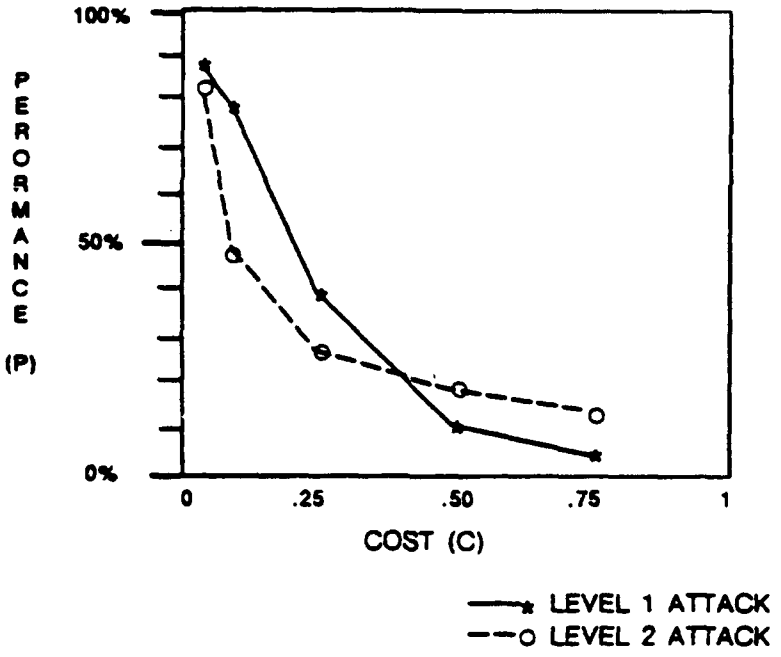


Figure A-2. Vulnerability Analysis Curve for Scenario *a* and Protocol Version *x*

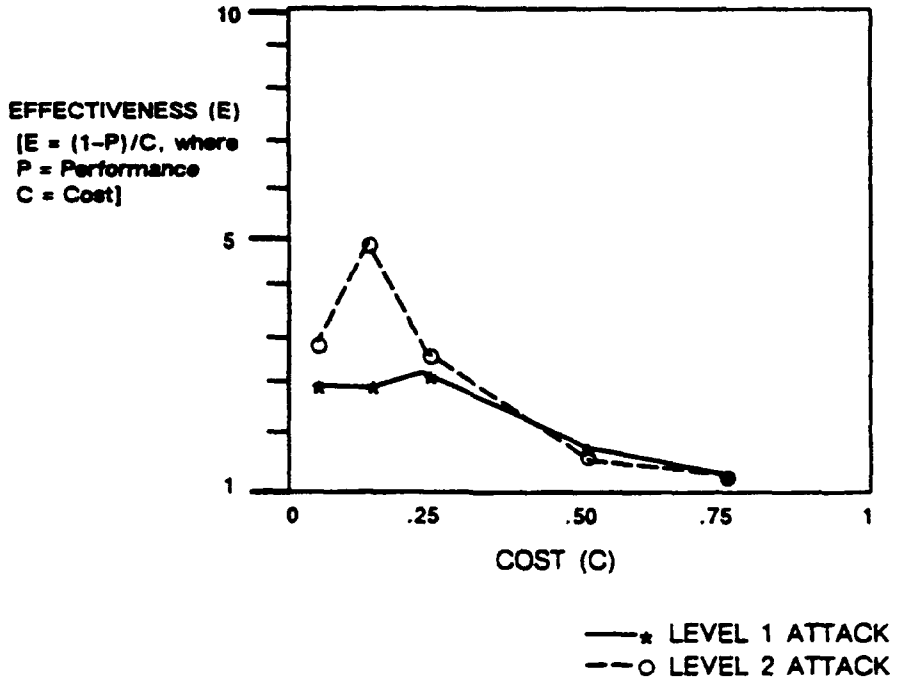


Figure A-3. Attack Effectiveness for Scenario *a* and Protocol Version *x*

A.5 CONCLUSION

This document has presented Part I (unclassified) of a larger report describing the EDMUNDS network environment profile, which is used to guide the development and analysis of the EDMUNDS algorithms and protocols. Benign network conditions were described, along with benchmark scenarios for protocol experimentation and a methodology for quantizing the performance of these protocols and their vulnerabilities.

The basic network environment, including transmission media, topology, traffic, and network dynamics, was first specified and then used to develop standard scenarios for evaluating and comparing the different protocols developed under the EDMUNDS program. Six benchmark scenarios were defined, and characterized by network size, nodal placement and identification, link assignment and connectivity, and jamming dynamics. Finally, a methodology including performance measures was developed for estimating protocol performance and vulnerabilities in benchmarks.

By estimating the characteristics, conditions, and requirements of multimedia networks in the future, the EDMUNDS network environment profile should also serve as a useful reference document for related network development programs.

Appendix B

HOW TO EXECUTE A NAPI+ SIMULATION

This section describes how to take a previously prepared NAPI+ simulation and use it to analyze the associated protocol. The experimenter must identify the appropriate executable simulation file, and must also prepare an input parameter file (.ef file) and a script file (.scr file).

B.1 SIMULATION FILE

The EDMUNDS project has implemented the following four protocols as NAPI+ simulations:

- EDMUNDS Baseline
- OSPF
- STIP2
- STIP3.

Each protocol simulation is in a single executable file that will run on a Sun SPARCstation* running SunOS. As released from SRI, the names of the latest simulation files are:

- bl_930802.ex: Baseline protocol, version 8 August 1993
- ospf_930802.ex: OSPF protocol, version 8 August 1993
- stip2_930802.ex: STIP2 protocol, version 8 August 1993
- stip3_930815.ex: STIP3 protocol, version 15 August 1993.

B.2 PARAMETER FILE

A parameter file must be prepared to provide input parameters for the simulation executable. The parameter file name must end with the suffix .ef, which stands for "environment file."

Each individual parameter specification consists of the name of the parameter, followed immediately by a colon (:), and ending with the value of the parameter. Lines that begin with the "pound sign" (#) character are treated with comments. Likewise, comments may be placed on each parameter entry line after the specification of the parameter value.

B.2.1 Environment Parameters

Regardless of which protocol is being simulated, there are several protocol-independent parameters that must be specified in every parameter file. These parameters are all listed in the example below. Keep in mind that the values given for each parameter are just examples, and will not be appropriate for all simulations.

```
env_duration: 60.0          # length of simulation (sec)
env_use_script: 1           # MUST be 1
env_script_name: "test.scr" # name of the script file
env_link_change_interval: 1.0 # interval for changing links (sec)
env_tg_streams: "6 3 .1 943" # src, dest, interval, bit_len

env_epoch_interval: 0.2     # env epoch interval (sec)
env_prop_delay: 0.001       # propagation delay on every link (sec)
env_settle_time: 5.0        # time to start generating traffic(sec)
env_seed: 1                 # random number seed

# parameters of the default link-probability estimator
env_gamma: 3.0              # smoothing constant for p(1)
```

*All product names mentioned in this document are the trademarks of their respective holders.

```
env_sigma: 0.8              # smoothing constant for ACK delay
env_t_mad_scalar: 0.2       # scalar for t_mad
env_t_scalar: 3.0           # scalar for p interval
env_max_t_scalar: 4.0       # scalar for maximum p interval

# parameters to control logging (usually 1/0 for on/off)
env_log_links: 1            # flag for logging of link dynamics
env_log_ete_d: 1            # flag for logging end-to-end delays
env_log_control: 1          # flag for logging tx of control packets
env_log_traffic: 1          # flag for logging tx of traffic packets
env_log_rx_control: 0       # flag for logging rx of control packets
env_log_rx_traffic: 0       # flag for logging rx of traffic packets
```

Although some of the parameters are self-explanatory, we explain the others below.

B.2.1.1 env_duration

The env_duration parameter indicates the length of simulated time (in seconds) the simulation should run.

B.2.1.2 env_script_name

The env_script_name parameter indicates the name of the script file, the file that contains the topology, and the link-dynamics script.

B.2.1.3 env_link_change_interval

The env_link_change_interval parameter indicates the time interval (in seconds) at which links are changed. This interval, multiplied by the tick number in the script file, gives the exact simulation time at which a link change in the script file will occur.

B.2.1.4 env_tg_streams

The value for the env_tg_streams parameter should be a comma-separated list of individual TG stream specifications. Four types of streams can be specified. The first two stream types, Simple and Special1, are nonbursty. The others, Special2 and Special3, are bursty streams.

In the specifications below, key words are given in bold-face type. A number or name as appropriate, should be substituted for all other items.

Where "dist" is called for, one of the following distributions can be named: constant (or const), uniform (or uni), or exponential (or exp).

Where "parm" or "parm_sec" is called for, a floating point number should be specified in seconds.

Where "seed" is called for, the number 1 should be specified.

Where "parm_bits" is called for, an integer number specifying a bit count should be given.

Where "init_state" is called for, 1 or 0 should be specified to indicate whether or not the stream should start off in a burst.

When "prob_remain" is called for, a floating point number should be specified. This number indicates the probability that the traffic stream will remain in the current burst.

- *Simple*: A constant-interval data stream. The format is <src, dst, interval, data_len_bits>. The following example specifies src-node 6, dst-node 3, a constant interpacket interval of 0.1 seconds, and a data length of 943 bits:

```
6 3 0.1 943
```

- *Special1*: The format is <src dst seed **interpkt** dist parm **pkt_len** dist parm_bits>. The following example specifies src-node 6, dst-node 3, seed 1, a uniform interpacket distribution with a parameter of 0.1 seconds, and a constant packet-length distribution of 943 bits:

```
6 3 1 interpkt uni 0.1 pkt_len const 943
```

- *Special2*: The format is <src dst seed **interburst** dist parm **interpkt** dist parm init_state **burst_len** dist parm_sec **pkt_len** dist parm_bits>. The following example specifies src-node 6, dst-node 3, seed 1, a constant interburst distribution with a parameter of 2.0 seconds, a constant interpacket distribution with a parameter of 0.1 seconds, an init_state of 0, a constant burst-length distribution of 0.5 seconds, and a constant packet-length distribution of 943 bits:

```
6 3 1 interburst const 2.0 interpkt const 0.1 0 \
        burst_len const 0.5 pkt_len const 943
```

- *Special3 (Markov)*: The format is <src dst seed **interburst** dist parm **interpkt** dist parm init_state prob_rem **pkt_len** dist parm_bits>. The following example specifies src-node 6, dst-node 3, seed 1, a constant interburst distribution with a parameter of 2.0 seconds, a constant interpacket distribution with a parameter of 0.1 seconds, an init_state of 0, a prob_remain value of 0.8, and a constant packet-length distribution of 943 bits:

```
6 3 1 interburst const 2.0 interpkt const 0.1 0 0.8 pkt_len const 943
```

B.2.1.5 env_epoch_interval

This parameter specifies the interval at which the environment's epoch processing executes. This processing includes (1) recalculation of the link-probabilities, and (2) the logging of all variables that have been specified for logging by the protocol.

B.2.1.6 env_prop_delay

This parameter specifies the propagation delay (in seconds) that is used for all links in the network.

B.2.1.7 env_settle_time

This parameter specifies the time (in seconds) at which the traffic generator will start all the traffic streams.

B.2.1.8 env_seed

This parameter specifies the random number seed (an integer) that will be used for some of the random number sequences in the simulation.

B.2.2 Baseline Protocol Parameters

The following example illustrates the parameters that must be set when a Baseline simulation is run.

```
bl_emax: 10.0                    # emax
bl_probe_interval: 0.2857        # probe packet interval
bl_use_indirect_link_acks: 1     # enable indirect links for acks
bl_use_indirect_link_updates: 1 # enable indirect links for updates
bl_use_forced_acks: 1            # enable "forced" acks
bl_update_interval: 2.0          # update pkt generation interval
```

B.2.3 OSPF Protocol Parameters

The following example illustrates the parameters that must be set when running an OSPF simulation. For an explanation of these parameters, consult the OSPF version 2.0 specification in Appendix C of this report.

```
ospf_minlsinterval: 0.25     # ospf default 5 secs (5.0)
ospf_rxmtinterval: 0.25      # ospf default 5 secs (5.0)
ospf_hellointerval: 0.5      # ospf default 10 secs (10.0)
ospf_pollinterval: 0.5       # ospf default 2 mins (120.0)
ospf_routerdeadhellomult: 4 # ospf default 4 * hellointerval
ospf_inftransdelay: 1.0      # ospf default 1 secs (1.0)
ospf_maxage: 3600.0          # ospf default 1 hour (3600.0)
ospf_maxagediff: 900.0       # ospf default 15 mins (900.0)
ospf_lsrefreshtime: 1800.0   # ospf default 30 mins (1800.0)
```

B.2.4 STIP2 and STIP3 Protocol Parameters

Refer to the STIP2* and STIP3† reports for an explanation of the parameters that must be set.

B.3 SCRIPT FILE

The script file contains the topology of the network and a script indicating the link dynamics. The script file used must be named via the env_script_name parameter in the parameter file.

B.3.1 Topology

The first number in the file indicates the number of nodes in the network.

The number of nodes (N) is followed by N locations, one for each node. The locations are not used by the simulation; however, the locations are logged into the log file, and are used by some analysis tools (including Netviz) to graphically display the topology of the network.

After the location entries, the connectivity matrix is specified, one line for each node. Each element in the matrix corresponds to a link. A link is enabled from node *i* to node *j* by the setting of element *i,j* to 1. All elements *i,i* should be set to 0 (nodes cannot have loopback links). In addition, the NAPI+ environment does not currently allow multiple parallel links per node, and the value of element *i,j* must be equal to element *j,i*. In other words, a maximum of one link is allowed between nodes, and if there is a link from node i to node *j*, then there must also be a link from node *j* to node *i*.

---

*Ogier, R.G., and D.S. Lee. 1993. *Secure Tactical Internet Protocol II (STIP2)* [in preparation].

†The report on STIP 3 is included in this report as Appendix E.

B.3.2 Link Dynamics

At the beginning of the experiment, all links are up. At any time during the experiment, the experimenter may schedule a link to go down or come up. To schedule a link to change state, first enter a line indicating at what tick-time the link should change. On the next lines, the experimenter should specify the links that should go down or come up. To specify a link, enter *i,j* for the link from node *i* to node *j*. To take the link down, enter the letter d; to bring the link up, enter the letter u. Link-state changes for multiple links may be listed after each tick.

Links may be brought down as early as tick 0 (t 0), the beginning of the simulation. The environment multiplies each tick number by the env_link_change_interval parameter to get the exact time at which the link should change state.

The simulation will always finish at the time indicated by the env_duration parameter, regardless of any link-state changes scheduled beyond that time.

B.3.3 An Example Script-File

The following is an example of a 3-node "string" network, where link 0,1 is brought down at tick 2, and at tick 5, link 0,1 is brought back up and link 2,1 brought down.

```
3
3 17
10 12
5 1

0 1 0
1 0 1
0 1 0
t 2
0,1 d
t 5
0,1 u
2,1 d
```

Appendix C

HOW TO ANALYZE A NAPI+ SIMULATION

This section provides information on how to use the EDMUNDS analysis tools to analyze protocol behavior in a NAPI+ simulation. Figure C-1 illustrates how the simulation analysis fits into the overall EDMUNDS simulation environment. Although many analysis tools were developed in the EDMUNDS project, the tools listed below proved to be the most useful. We explain these tools in the subsections that follow.

- ScriptGen
- DoStipBatch
- plot_ete
- plot_links
- Netviz and the nv filters
- Performance
- p2
- p3.

All of the above tools operate on the data contained in the log file generated by a NAPI+ simulation.

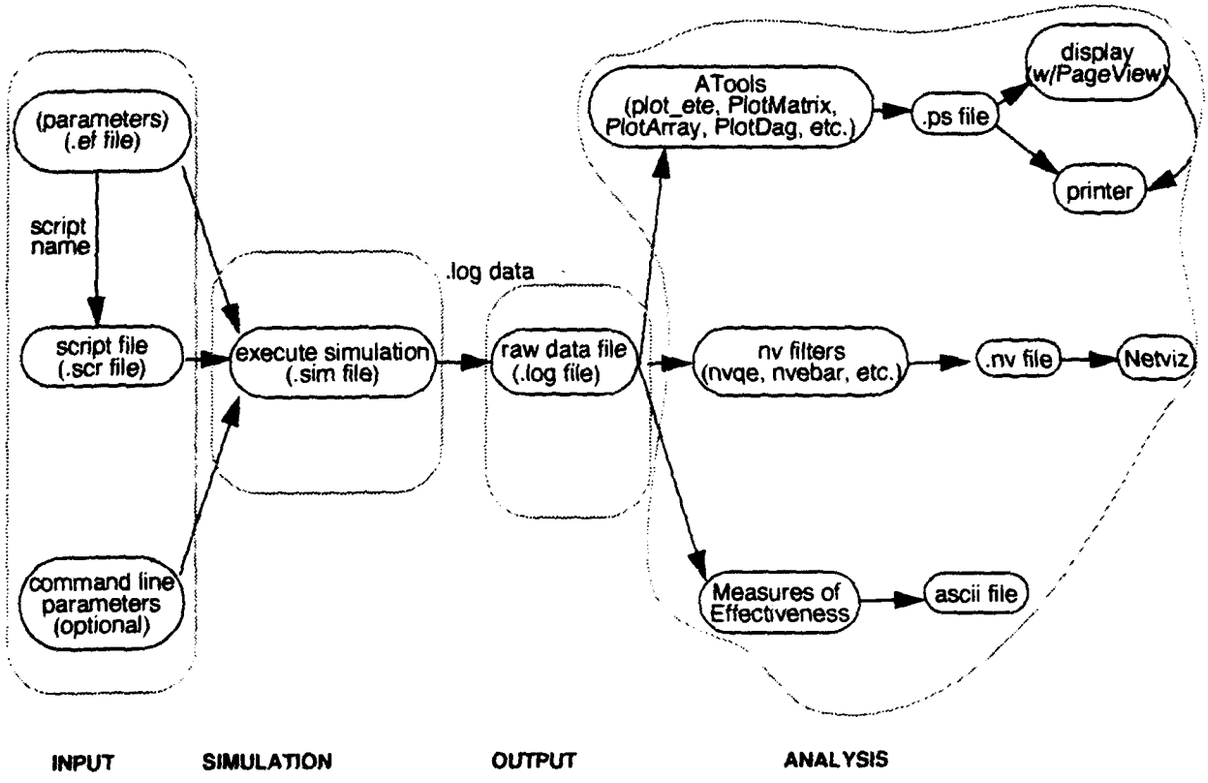


Figure C-1. The EDMUNDS Simulation Environment

C.1 DoStipBatchScriptGen

ScriptGen, a tool written in AWK, allows the user to generate the node positioning, link connectivity matrix, and link dynamics specified in the .scr file that is used by the simulation. For instance, rather than entering the connectivity matrix into the .scr file by hand (e.g., in a 10 × 10 matrix of 1 s and 0 s), ScriptGen helps to automate this process and only requires the user to create an input file using simple commands to describe nodal location and connectivity between two nodes. The following eight commands can be used to specify a scenario.

- NetSize <number of nodes>—mandatory command at the beginning of the file
- Interval <seconds>—link change interval as specified in the .ef file; mandatory before the Time command
- Position <node number> <x location> <y location>—optional before Start command; nodes are numbered from 0 to Netsize-1, but can be listed in any order
- Capacity <tail node> <head node> <integer bits per second>—optional before Start command
- Start—mandatory; causes header information to be output
- Time <seconds>—optional after Start command; each Time command must specify a time that is an even multiple of the Interval and is greater than the previous Time
- Up <tail node> <head node>—anywhere in the file; used to specify initial connectivity and subsequent link dynamics after the Start command
- Down <tail node> <head node>—anywhere in the file; used to specify link dynamics after the Start command.

The following is an example of an input file (e.g., GENSQUARE) to ScriptGen; this file describes a four-node-square network with link 0<->3 down at Time 3.

```
NetSize 4
Interval 1
Position 0 1 0
Position 1 1 1
Position 2 3 1
Position 3 3 0
Up 0 3
Up 0 1
Up 1 0
Up 1 2
Up 2 1
Up 3 0
Up 3 2
Start
Time 3
Down 0 3
Down 0 1
```

This tool is invoked by

```
ScriptGen GENSQUARE
```

The output should be piped into a ".scr" file for use by the simulation. The output of this ScriptGen invocation is as follows:

```
4
1 0
1 1
3 1
3 0
0 1 0 1
1 0 1 0
0 1 0 1
1 0 1 0
t 1
t 2
t 3
0 3 d
0 1 d
```

C.2 DoStipBatch

DoStipBatch is a software tool that enables the experimenter to automate the execution of a number of simulations. This tool helps to ensure reliable sequencing of the experiment protocol parameter settings, topological changes, traffic variations, and nodal behavior; and allows for easy repetition of the same or similar experiments at future times. DoStipBatch is particularly useful in determining the sensitivity of particular protocol parameters that require a large number of simulations to be run, and in running a series of overnight simulations.

This software tool is a Perl script that takes as input a file that contains a description of each simulation to be run within the particular batch job. Basically, the input file contains the name of the simulation executable, the environment file (i.e., the .ef file), the script file (i.e., the .scr file), and the environment file parameter settings that are different from those in the .ef file. For each simulation specified, DoStipBatch executes the simulations and outputs end-to-end delay measurement plots from plot_ete and the files .perf, .rep, and .dump, which are described below.

The following is an example of an input file (e.g., TEST) to DoStipBatch:

```
:LABEL TEST(EP.15)
stip3_930729.ex -ef stip3 -env_script_name test.scr \
-env_tg_streams "6 3 .2 943"

:LABEL TEST(EP.1)
stip3_930729.ex -ef stip3 -env_script_name test.scr \
-env_tg_reams "6 3 .2 943" -s3_epsilon .1

:LABEL TEST(EP.05)
stip3_930729.ex -ef stip3 -env_script_name test.scr \
-env_tg_streams "6 3 .2 943" -s3_epsilon .05
```

This example contains three simulation runs. The difference between each simulation is the STIP3 parameter setting for s3_epsilon. In the first case, the value is as specified in the default stip3.ef environment file. In subsequent runs, the values are 0.1 and 0.05. Thus, TEST is examining the sensitivity of the protocol to s3_epsilon.

The first line of each simulation description enables the user to identify the simulation—i.e., TEST(EP.15) or TEST(EP.1)—and can be of any length; this identification is automatically placed on the output of the plot_ete tool. In the above example, the protocol executable is

stip3_930729.ex; and stip3.ef is the associated environment file. Stip3.ef is a default .ef file any of whose parameter settings can be changed for each simulation. These changes are made by simply adding the parameter name to the simulation description, preceding the name with a dash (-) and following it by the desired setting. Thus, in the above example, s3_epsilon is set to the value given in stip3.ef for the first simulation; in the two subsequent simulations, s3_epsilon is set to 0.1 and then to 0.05. In this example, the .scr script file and the traffic streams are identical for each of the three simulations. There is no limit to the number of simulations executed using the DoStipBatch command. It should also be noted that not all simulation executables and .ef files are the same, as in this example. This tool is invoked as follows:

```
DoStipBatch TEST
```

The output files to DoStipBatch are preceded with the name of the input file. The following briefly describes the output from DoStipBatch for the example above:

- TEST.rep—a summary of stream information (e.g., number of streams, stream source and destination, and traffic rate)
- TEST.perf—the output of performance measures from the Performance tool
- TEST_n.dump—simulation input parameters, node location and neighbors, and statistics for each node (e.g., the percentage of packets received, the number of packets dropped, the number of packets sent from all other nodes) for each simulation (for this example, n = 0, 1, and 2)
- TEST_n.ef—the resulting environment file for each simulation, based upon settings defined in the input file to DoStipBatch (for this example, n = 0, 1, and 2)
- TEST_n.ps—the topology plot for each simulation (for this example, n = 0, 1, and 2)
- TEST_n.log—the output log file for each simulation (provided any of the log parameters were set to 1; for this example, n = 0, 1, and 2)
- TEST_n.log_ete_s_d.ps—the output from plot_ete for each source (s) and destination (d) stream (for this example, n = 0, 1, and 2; s = 6; and d = 3).

C.3 plot_ete

plot_ete is a software tool that generates a plot of the end-to-end delays of all the traffic packets sent during a simulation. This tool takes as input a .log file created by running a NAPI+ simulation. As output, plot_ete invokes the Sun utility PageView to display the end-to-end plot on the workstation screen. In addition, plot_ete puts the plot in a PostScript file that can then be printed on a PostScript-capable printer. The PostScript file or the hard copy of the plot from the printer can be used as a permanent record of the simulation behavior.

The following is an example of how plot_ete would be invoked using the log file foo.log, which contains end-to-end data for a traffic stream between node 6 and node 3.

```
plot_ete foo.log 6 3
```

Any given log file may have end-to-end data for more than one traffic streams. If such is the case, plot_ete is invoked for each individual stream (src/dst pair) that is to be analyzed.

The end-to-end plot is annotated with the mean end-to-end delay, the delay variance, the number of traffic packets transmitted and received, and the overall reliability of the associated stream. An example of the output from plot_ete can be seen in Figure C-2.

plot_ete is used to provide an overview of network behavior during an entire simulation; it is particularly useful in examining behavior as a result of the link dynamics contained in the script file. However, plot_ete is not useful in showing which routes traffic packets took during the simulation. For a graphical illustration of packet flow, Netviz is used.

C.4 plot_links

plot_links is a software tool that displays link usage for a given node. It is useful in determining the flow of traffic from a node: for instance, one can use this utility program to see the degree of spreading of traffic.

This tool takes as input a .log file created from running a simulation. As output, plot_links invokes the Sun utility PageView to display traffic, updates, acks, and probe packets transmitted over each link for a given node. In addition, plot_links puts the plot in a PostScript file that can then be printed on a PostScript-capable printer.

The following is an example of how plot_links is invoked via the log file foo.log, which contains the end-to-end data for traffic streams between nodes 0 and 7 and nodes 0 and 8, and control traffic (link usage is displayed for node 0):

```
plot_links foo.log 0
```

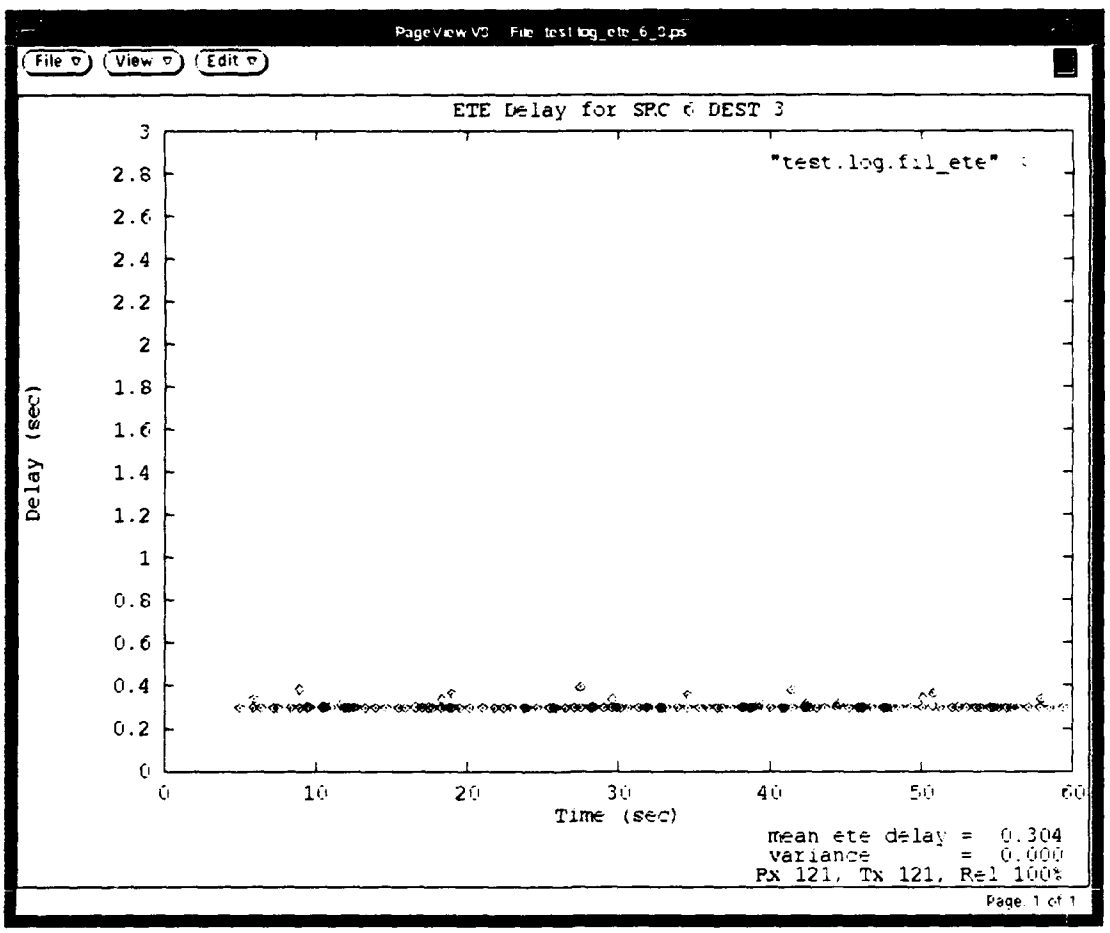


Figure C-2. Example of plot_ete Window

An example output display is shown in Figure C-3. For each link emanating from node 0, the display shows the aggregate traffic packets (T) for all streams, in addition to the traffic packets (T0...Tn) for each stream separately. The display also shows update (U), ack (A), and probe (P) packets transmitted over each link for the given node. The y axis identifies the links associated with the node—with each increasing y-axis number representing the link between the specified node and the next-highest-numbered neighbor node.

C.5 NETVIZ AND THE nv FILTERS

Netviz is a graphical network-display tool used to assist network developers and managers in analyzing the behavior of network algorithms. Written in C++, Netviz is based on the X Window System and the OSF-Motif user interface toolkit.

To update the display, Netviz reads from a file consisting of simple, network-oriented display commands. These files are referred to as *nv*. Typical display commands are used to display bargraphs, node text, link colors, and packet transmissions. Analysis can be performed either in real time as the network generates the data, or off line, to review past experiments. Figure C-4 illustrates the steps taken during a Netviz analysis.

In the section which follows, we describe the filter tools used to translate the data in the simulation log files to the nv-file format. We then describe the Netviz tool itself.

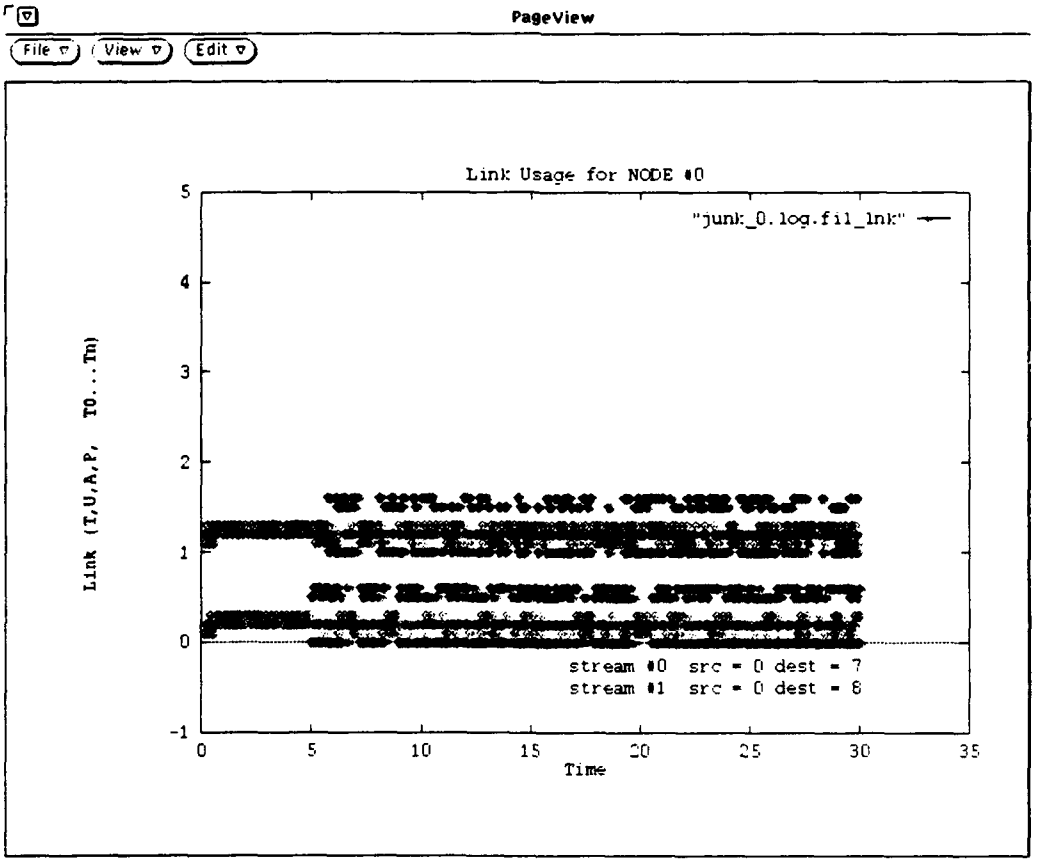


Figure C-3. Example of plot_links Window

C.5.1 nv Filters

The data contained in a NAPI+ log file must be filtered to create a customized version of what the experimenter would like to see animated in Netviz. The log file contains a wealth of detailed protocol-dependent and protocol-independent data that must be mapped into the context of the visual features provided by the Netviz network display. For example, the log file records the transmission of all the packet types in a simulation, such as TRAFFIC, UPDATE, ACK, and PROBE. The experimenter may decide to display each packet transmission using a color unique to each packet type; so an appropriate nv filter must be selected or created. Alternatively, the experimenter may wish to highlight different traffic streams and therefore, an nv filter should be selected which assigns unique colors to packet transmissions according to the final-destination of the packet being transmitted.

Various nv filters were created and should be available as part of the Netviz release. All of these filters were written in Perl, a text-manipulation programming language. To execute the filters, the users must have access to Perl on their workstations. Some of the filters are described below:

- nvebar/nvqe: shows the values of simulation variables such as expected delay (ebar), queue size (q), and average queue size (qbar), on a per-node basis.
- nvdots/nvdots.qbar: computes traffic volume per node in a rolling time window and illustrate the volume by varying the size of the node. In addition, nvdots.qbar illustrates qbar for each node.
- nvpkts/nvstrms/nvqstrms/nvqestrms: uses a unique color for each packet type transmitted. In addition, nvqstrms and nvqestrms illustrate queue sizes.

All of the EDMUNDS nv filters take a log file as an argument, and print the resulting nv data. Thus, an nv filter can be invoked to create an nv data file, as shown in the following example:

```
nvstreams test.log > test.nv
```

In this example, the experimenter would then load the test.nv into Netviz.

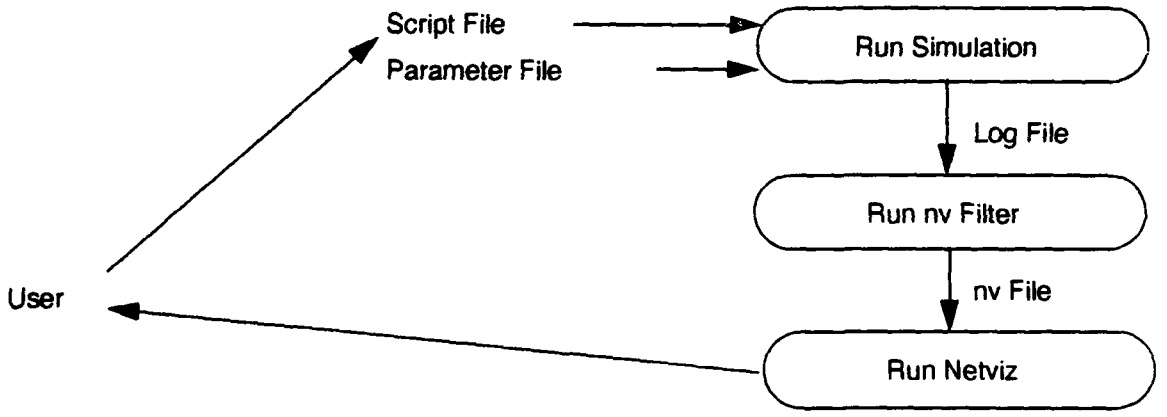


Figure C-4. Steps for Netviz Analysis

C.5.2 Using Netviz

Netviz animates networks by displaying filled-in circles for nodes, lines for links between nodes, and moving polygons for packets traversing the links. Figure C-5 shows an example of how nodes, links and packets appear in a Netviz display. See Section C.5.3 for a detailed description of the animated packets.

Netviz presents a horizontal menu bar at the top of the tool for activating infrequently used features. This menu bar presents the Main and the View menus. These menus are selected by left-clicking on the menu title. In addition, at the bottom of the tool window are several buttons and editable-text windows that allow the user to step through the animation. Below are descriptions of the commands in each menu and of the animation execution controls.

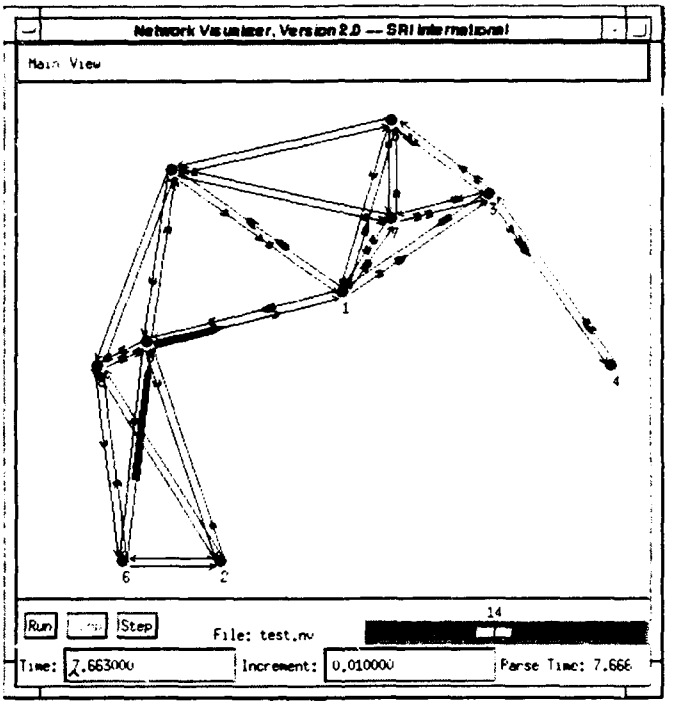


Figure C-5. Netviz

C.5.2.1 Netviz Main Menu

The Netviz Main button is located in the top left corner of the Netviz screen on the horizontal menu bar. There are four options available in the Netviz Main menu: File, Rewind, Info, and Quit.

The File command gives the user a popup window used for selecting/inputting the name of an nv file for driving the Netviz display. The File popup window presents an interface for navigating (browsing) through the file system to find an appropriate nv file to load, including a menu of available files in the current directory. By convention, Netviz files are suffixed with .nv, so the initial name filter (specified in the File popup) is set such that the file name menu presents only files that end in .nv. Selecting the File command also resets all the run-time parameters to their default values (see Section C.5.2.3 below on Netviz run-time control).

The Rewind command rewinds the current selected nv file to the beginning so that the same file can be animated again. The Rewind command is also appropriate for rewinding when the same file has been refilled with new experiment/animation data. Rewind does not reset run-time parameter settings which are already in place.

The Info command presents information about Netviz: the version number, the compilation date, and an acknowledgment to DARPA (ARPA) and Rome Laboratory.

The Exit command allows the user to quit Netviz.

C.5.2.2 Netviz View Menu

The Netviz View button is located to the right of the Netviz Main button. There are five selections in the View menu: Redraw, Zoom in, Zoom out, Pan, and Original.

The Redraw command refreshes the Netviz screen.

The Zoom in command magnifies the current Netviz display, while the Zoom out command shrinks it.

The Pan command allows the user to pan (slide) the display, so that user can focus on different parts of the animated network. This is especially useful when the user has zoomed in and needs to adjust the viewed area. After selecting Pan, the user pans by left-clicking on a point in the display that will become the new center of the viewed area.

The Original command restores the Netviz display to its original state before any Pan or Zoom operations were performed.

The combination of Pan, Zoom, and Original allows the user to focus on the Netviz display from different perspectives. When a new file is selected for animation, or the current file is rewound, the display perspective is automatically reset to Original.

C.5.2.3 Netviz Run-Time Control

At the bottom of the Netviz screen are buttons and parameters for controlling the run-time execution of an animation. The text parameters are represented and changed in editable-text windows. The text windows have default values that are used unless they are edited (overridden) by the user.

The Run button tells Netviz to start or continue the animation. Run runs through the entire nv file and stops after the last command in the nv file.

The Stop button tells Netviz to stop the animation.

The Step button tells Netviz to advance one step in the animation. One step in the animation equates to one increment in time, as set in the Increment box. Step allows the user to step through a simulation in slow motion.

The Time button shows the current time of the animation. The default value for Time (0.0) is set before the animation starts. However, the user can choose to start or warp to any time in the simulation by editing the Time box. The user can click the left mouse button anywhere in the Time box and edit the number to contain a new value to which Netviz will warp upon starting (or restarting) the animation with Run. Note that the Del key is used to delete forward, and the Back Space key is used to delete backward. Warping is useful when the user wishes to skip to a future

time in the animation without having to view the uninteresting parts of the animation. When the user warps (by changing Time to some point in the future), no new events are animated until Netviz parses through the nv file and reaches an event at or after the specified time.

The File display window shows the name of the nv file currently being animated. The File display window does not accept user input.

The Increment button allows the user to choose a time increment at which Netviz advances its animation clock. The larger the increment, the faster, and therefore coarser, the animation runs. The default value for Increment is 0.01 seconds. Like the Time box, the user can change Increment by selecting the Increment box with the left mouse button and editing a new value.

The Slide Bar to the bottom right of the Netviz display screen adjusts to some degree how fast the animation is displayed. The display speed can be adjusted by dragging the bar to the left (slower) or to the right (faster) with the left mouse button.

The Parse Time display window shows the time of the individual animation commands in the nv file through which Netviz is parsing (reading). Parse Time may or may not have the same value as Time, the current time of the animation, because the animation advances at Increment time, but the Netviz commands in the nv file can have arbitrary time. The Parse Time display window accepts no user input.

C.5.2.4 Typical Sequence of Operations

The following is a normal sequence of operations when Netviz is used for animating network simulations:

1. To invoke the netviz tool from the shell, type in the command
   ```
   netviz myfile.nv
   ```
   or simply
   ```
   netviz
   ```
2. The Netviz tool window appears on the screen. If no file name was given at the shell command line, select File in the Main menu to input/select an nv file.
3. If desired, adjust Time, Increment, and the Slide Bar. You will have to adjust the values for these parameters to learn what is appropriate for any given animation.
4. Select the Run or Step button to view the animation.
5. If the current view of network is not satisfactory, select Stop to pause the simulation, select the Pan and/or Zoom commands to adjust the view, and then resume animation by selecting Run or Step.
6. The animation can also be stopped (or paused) at any time to adjust the animation execution parameters Interval and Time.

C.5.3 Netviz Animated Events

The following is a more detailed description of the Netviz display when a packet is transmitted on a link.

First, we define the following terms:

- propagation_delay: the time necessary for a signal to travel from one node to the next (independent of packet length).

- transmission_delay: the elapsed time between the start of transmission of the first bit of a packet and the end of transmission of the last bit. Specifically, this value is size_of_packet/link_speed. For example, transmitting a 1,000-bit packet on a 10,000-bit/s link results in a 0.1 second transmission delay.
- start_xmt_time: the time at which a link begins to transmit the first bit of a packet.

Netviz displays packet movement by drawing a snapshot of a packet at each time increment. By default, this increment is set to 0.01 s and is displayed in the user interface. At each time increment, the packet is indicated by a closed polygon with a head and a tail position on the link it is traversing:

When and where the packet is overlaid on the link is governed by the following equations. Note that in these equations, "head" and "tail" are unitless, and range from 0.0 through 1.0. Given a directional link from X to Y, head is a percentage, and indicates how far across the link the head of the packet should start. Likewise, tail is a percentage, and indicates where the packet should end.

The head and tail are computed as follows:

- head = (current_time - start_xmt_time) / propagation_delay
  This essentially computes how far along the link the first bit of the packet should be at the current time. Note that head can compute to be greater than 1.0, meaning that some part of the head of the packet has already been received by Node Y. If head is greater than 1.0, the head remains "parked" at the end of the link (at the edge of the receiving node).
- tail = (current_time - (start_xmt_time + transmission_delay)) / propagation_delay)
  Tail is computed similarly to head, but indicates where the tail should be drawn. Note that tail could compute to be negative, meaning that the some part of the tail of the packet has not yet been transmitted out onto the link. We limit tail to a minimum of 0. Thus, the end of the packet will be drawn at the beginning of a link, unless the snapshot time (current_time) is in the propagation delay interval (i.e., between [start_xmt_time + transmission_delay] and [start_xmt_time + transmission_delay + propagation_delay]).

C.5.3.1 Regarding Propagation Delay

When we first started using Netviz, we found that the depiction of the flow of packets was choppy: a packet would flash over the full length of the link for an instant, and then disappear. The user would rarely or never see the head or the tail flow across the link, and thus could never see two packets on the link at the same time. While the implementation was correct, the resulting display was frustrating. Our solution was to enable the nv filters (which translate the data from the log file to the nv for Netviz) to "lie" about the propagation delay when appropriate. That is, some nv filters were made to output propagation delays of 0.18 instead of the actual (and more realistic) delay used by the simulation (e.g., 0.001 s). The bigger the propagation delay, the slower the packet motion, and the more smoothly the packet seems to flow through the links. This new behavior makes it much easier for viewers to get an overall perspective of packet flows in the network.

The downside of this solution is that the same packet is displayed on multiple links at the same time. Specifically, the nv filter propagation delay has been exaggerated, but the start_xmt_time of the next hop for that packet is still accurate. The likely effect is that the receiving node will seem to transmit a packet on the next hop before the p is displayed as fully received, or even before the head is received—unless the packet is queue long time. We may even see a packet exiting from several nodes at the same time.

The bigger the propagation delay, the more such "overlap" occurs. An example of this overlap is detailed below, with propagation delay set to 0.18 for all links (the real propagation delay was 0.001); packet length at 1000 bits, and all link speeds at 10,000 bits/s. Recall that the total time it takes for a packet to transverse the link (or the time for the last bit of the packet to be received) is [transmission_delay + propagation_delay]; and the time at which a packet is fully received by the receiving node is [time_done = start_xmt_time + transmission_delay + propagation_delay].

1. Scenario: A packet is transmitted from node 1 to node 2, then on to node 3.
2. Events:
   - start_xmt_time at node 1 = 0.000 (accurate)
   - Packet received by node 2 at 0.101 (0 + 0.1 + 0.001)
   - time_done = 0.28 (0 + 0.1 + 0.18)
   - start_xmt_time at node 2 = 0.101 (accurate)
   - Packet received by node 3 at 0.202 (0.101 + 0.1 + 0.001)
   - time_done = 0.381 (0.101 + 0.1 + 0.18)
3. Observation: The time window for the animation of the packet transferring from node 1 to node 2 overlaps the time window for the transfer from node 2 to node 3. As a result, the user sees (in the Netviz display) packets emerging from both node 1 and node 2 between time 0.101 and 0.280. These two packets actually represent the same packet in the simulation, which may confuse the user.

As previously stated, the solution to fixing a choppy display is to "optionally lie" about the propagation delay. If the user desires the correct behavior (using the propagation delays of the simulation), the short term but somewhat tedious solution is to edit the nv filter to output 0.001 (or whatever propagation delay was set for the experiment) instead of 0.18. After editing the nv filter, the user must run the nv filter again, but need not rerun the simulation.

C.6 PERFORMANCE TOOL

The Performance tool is a Perl script that computes the Measures of Effectiveness (MOE) as described in the EDMUNDS Network Environment Profile document (included in Appendix A of this report). Performance takes a simulation log file as input. In its simplest form, Performance is invoked as follows (where file.log is the name of the log file):

```
Performance file.log
```

An example of the output is show below:

```
S: 6->3 gen N: 116 T: 2.2311 D: 0.3042 R: 1.00 F: 0.00 M: 0.9914
Total N: 116 T: 2.2311 D: 0.3042 R: 1.00 F: 0.00 M: 0.9914
Performance 0.999927
```

For an explanation of the output, refer to the EDMUNDS Network Environment Profile document.*

C.7 p2

p2 is a general purpose perl utility that takes pairs of numbers from an input file as abscissae and ordinates of a graph, and displays a curve. The ordinate values need not be in any order; however, when the line points and line options (described below) are used, it is recommended that the ordinate values be listed in increasing order, because points are connected in the order they appear in the input file. A number options are available to title the graph, label the axes, and specify the style of displayed values. These include

- -x<str>, to specify the ordinate axis
- -y<str>, to specify the abscissa axis
- -t<str>, to specify a title for the graph
- -style <points|dots|impulses|linepoints|lines>, to specify display and connection style; the latter two options connect the points on the graph in the order they appear in the input file.

The following is an example of how p2 is invoked via the input file <filename> which contains pairs of numbers to be graphed:

```
p2 -x"X TITLE" -y"Y TITLE" -t"MAIN TITLE" -style dots filename
```

C.8 p3

p3 is similar to p2, except that p3 displays two curves rather than one, and takes as input two files, each containing pairs of numbers (i.e., abscissae and ordinates of a graph). Ordinates of both files must be the same. As in the case of p2, the ordinate values need not be in any order; however, when the linepoints and line options are used, it is recommended that the ordinate values be listed in increasing order, because points are connected in the order they appear in the input file. The available style options are the same as those for p2. The following is an example of how p3 is invoked, using the input files file1 and file2 with the corresponding output display shown in Figure C-6.

```
p3 -x"X TITLE" -y"Y TITLE" -t"MAIN TITLE" -style linespoints file1 file2
```

C.9 GUIDELINES

Each of the discussed analysis tools performs significantly different tasks. ScriptGen and DoStipBatch are tools to help facilitate the setting up and execution of the experiments. In particular, once a scenario has been defined, ScriptGen provides a simple way to create the script file for the simulation. DoStipBatch, on the other hand, provides a convenient way of automating the process of executing a large number of simulations; in addition to being helpful for running jobs overnight, DoStipBatch should be used to set up experiments designed to study, for instance, the sensitivity of a particular protocol parameter.

Once the simulations have been executed, a number of jobs are used to analyze the experiment. Plot_ete is useful for seeing, at a glance, the ETE delay throughout the duration of the experiment. Plot_ete is also useful in roughly quantifying the behaviors caused by scheduling link dynamics and traffic packet generation. Netviz is useful in monitoring the routes that packets take

*Lee, D.S., D.A. Beyer, and R.G. Ogier. 1992. *EDMUNDS Network Environment Profile: Benign and Adversarial Conditions, and Benchmark Scenarios (Part I)*, ITAD-8558-TR-91-23, SRI International, Menlo Park, California (May).

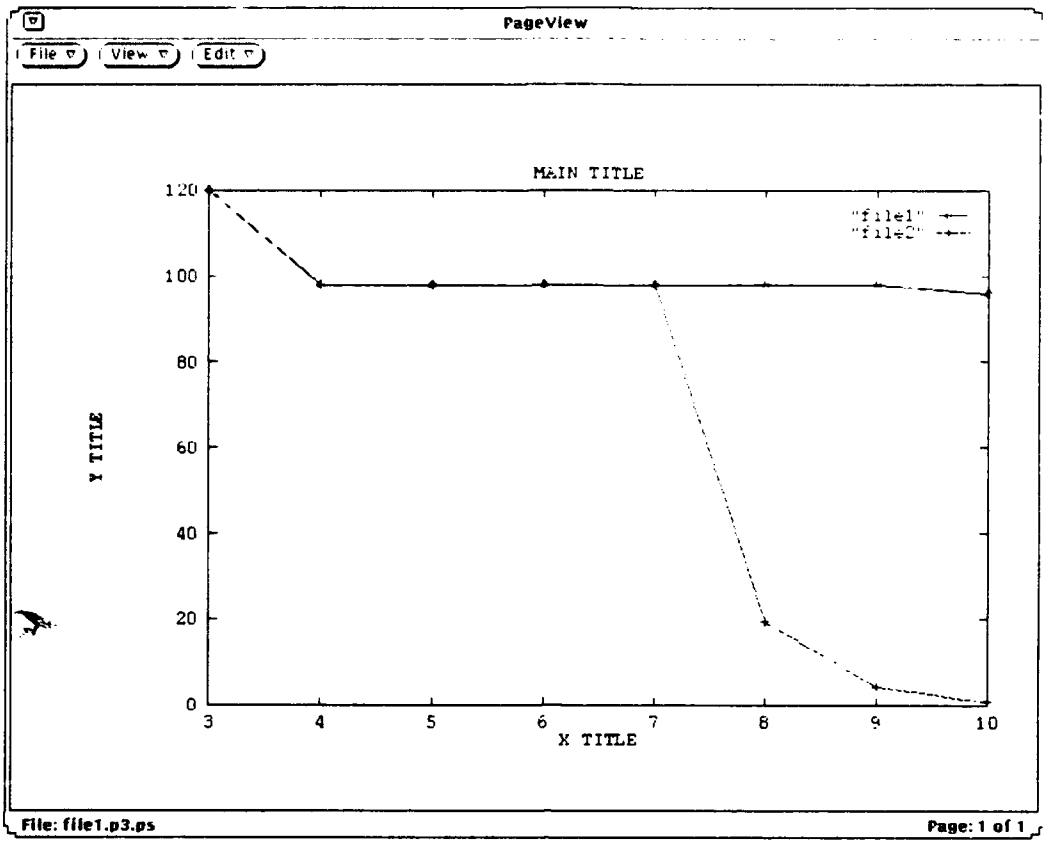


Figure C-6. Example of p3 Window

as they are forwarded through the network. Together, plot_ete and Netviz provide an effective method for debugging and tuning protocol behavior at a low level. Plot_links is also used to examine the behavior of the protocol at a low level, and is often used when Plot_ete and Netviz do not provide the information needed to analyze the protocol. For instance, Netviz can be used to show link usage by different packets at a given snapshot; Plot_links, on the other hand, provides this information for the entire experiment at a glance. Performance, however, is a much more rigorous analysis tool, and is much more appropriate for analyzing batches of simulation runs.

Appendix D

HOW TO IMPLEMENT A NETWORK PROTOCOL USING THE NAPI+ ENVIRONMENT

HOW TO IMPLEMENT A NETWORK PROTOCOL USING THE NAPI+ ENVIRONMENT

D.1 BACKGROUND

NAPI+ is a specification for an object-oriented programming environment for network protocol development. The NAPI+ environment is an implementation of the classes and functions in the NAPI+ specification and includes many classes and functions that are not visible in the specification. The specification consists of three kinds of classes: (1) classes that provide protocol-independent services that are useful when implementing a full simulation or packet-switch implementation; (2) protocol-independent base classes from which the protocol developer must derive protocol-dependent classes; and (3) an assortment of minor functions that the protocol module must implement in order to glue the environment to the protocol.

To create a NAPI+ simulation, the developer must implement in C++ a protocol module that is compliant with the NAPI+ specification. It is essential that the developer have a solid understanding of the C language. Further, implementation of the protocol will be much easier if the developer has good experience with C++; however, the developer should be able to get by with only a smattering of C++ knowledge if the implementation process starts with an existing protocol module and proceeds with conversion or porting.

The protocol developer needs to provide protocol-specific objects and functions, as described in Subsection D.2 and Subsection D.3. Subsections D.4- D.9 describe the support services provided by NAPI+ to help the protocol developer manage tasks such as manipulating hello/update packets, accessing packet queues, and retrieving simulation parameters from a configuration file. Subsection D.10 walks through an example of how environment and protocol objects interact to process an event. Finally, Subsection D.11 describes how to compile NAPI+ simulations.

Note that in the discussion that follows we intentionally do not list the arguments for some functions, in cases where we feel the actual arguments are not important in the context of the surrounding text. For a full specification of all the NAPI+ class functions and their arguments, refer to the NAPI+ header (.h) files in the source code release for the NAPI+ environment module. Figure D-1 summarizes the protocol-dependent classes that the protocol developer must implement.

D.2 REQUIRED PROTOCOL CLASSES AND FUNCTIONS

To use the NAPI+ simulation environment, the protocol developer must define and implement several protocol-dependent C++ classes that are derived from classes in the NAPI+ specification. Defining a required derived class consists of creating a C++ class specification and implementing the class member functions and data structures to execute the protocol-specific behavior. In addition, the protocol developer has complete freedom to create additional functions and classes to implement the required classes and functions in the protocol module. This section describes the classes that must be defined and implemented.

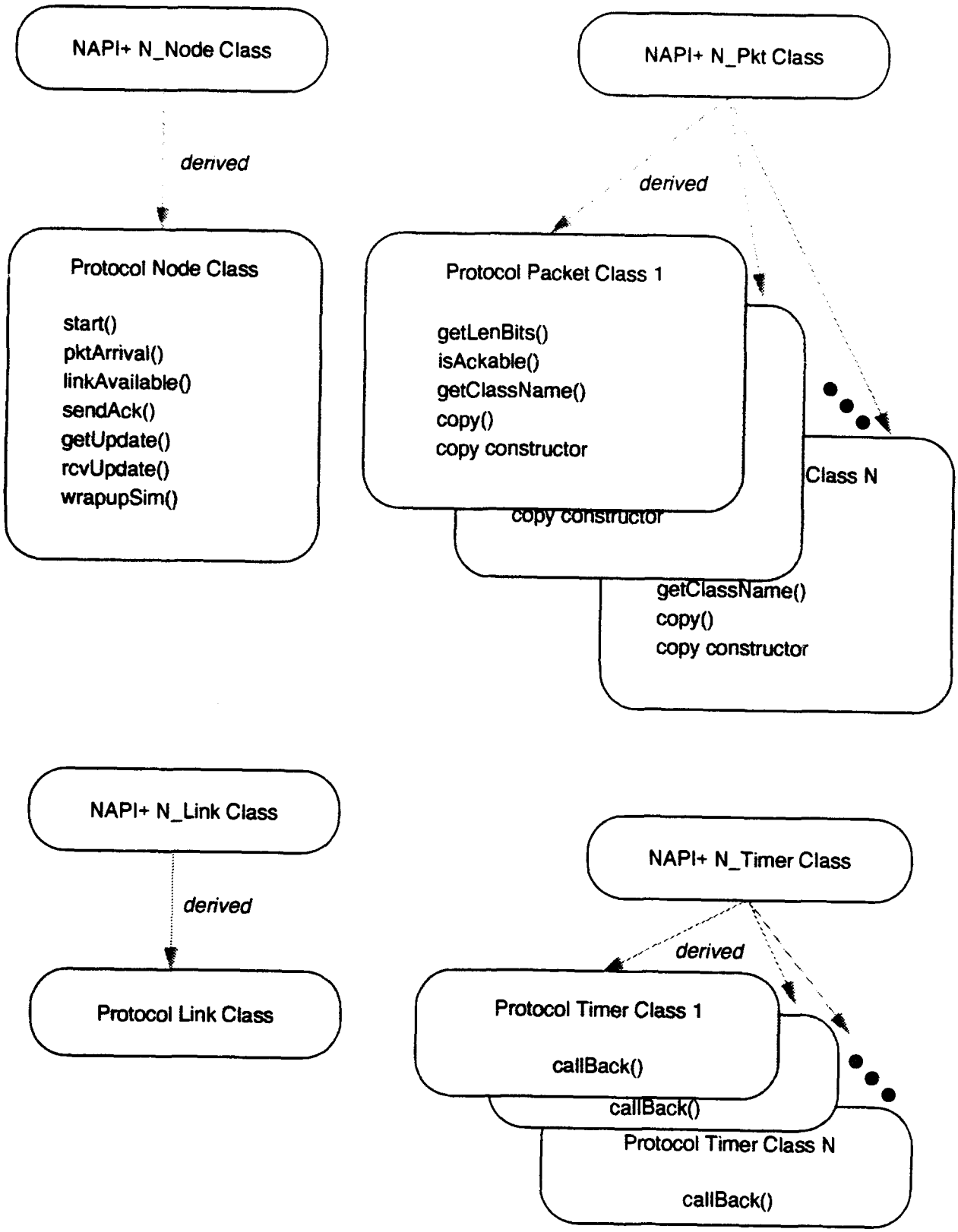


Figure D-1. Required Protocol Classes and Functions

Constructor and destructor functions are required for every class created in the protocol module, but the behavior of these functions is protocol specific and depends on the data structures contained by each class. We will not describe the required constructors and destructors here, except to say that, in general, class destructors must destroy all the objects that are created in associated constructors.

Keep in mind that most examples given in this section are taken from the NAPI+ implementation of the EDMUNDS Baseline protocol. For the sake of simplicity, the examples we present may not reflect the exact behavior of the Baseline protocol.

Also note that all classes in the NAPI+ interface are prefixed by the string N_. Thus when you encounter such a class name in this document, you can infer that the definition of the class is in a NAPI+ header file, and the implementation of the class is provided in the environment module. The exception is that some of the functions of the N_Env class need to be implemented by the protocol developer; however, these exceptions will be discussed later in this document.

D.2.1 Protocol Node Class

The developer must implement a protocol-specific node class, which must be derived from the base class N_Node. The NAPI+ specification does not currently handle multiple protocols or protocol layers. Thus, we incorporate protocol behavior into the node class, rather than creating separate node and protocol classes. For the purposes of discussion, we shall refer to the protocol node class as PNode. Seven mandatory protocol-specific functions are required for a PNode:

- start()
- pktArrival()
- linkAvailable()
- sendAck()
- rcvUpdate()
- getUpdate()
- wrapupSim().

Following is an example of a PNode:

```
/* protocol Node class is derived from the NAPI+ N_Node class */
class Node : public N_Node {
public:
    /* mandatory protocol-specific functions must be defined here */
    void    start();
    void    wrapupSim();
    void    pktArrival(N_Pkt*, int arrival_mode, N_Link*);
    N_Pkt*  linkAvailable(N_Link*);
    void    rcvUpdate(int reporting_node, float time,
                      int var_id, int index, int value);
    int     getUpdate(int var_id, int index);
    /* other functions */
private:
    /* data structures */
};
```

The requirements of each function follow.

D.2.1.1 start()

This function dictates the behavior of a protocol node at the start of a simulation. The start() function is called by the environment module when it is time to start executing the protocol. There are no explicit requirements of this function; however, it should create all the protocol node's substructures and initialize its variables. The start() function should also schedule the very first events to be processed by the protocol node. For example,

```
void Node::start() {
    /* create protocol node class substructures */
    _v = new N_ArrayFloat(getNumOutLinks(), "v", 0);
    _queues = new N_PriorityTimeQueue*[getNumOutLinks()];
    for (int i = 0; i < getNumOutLinks(); i++) {
        _queues[i] = new N_PriorityTimeQueue();
    }
    /* create timer and schedule first interrupt */
    _update_timer = new UpdateTimer(this);
    _update_timer->set(timeout, NULL);
}
```

D.2.1.2 pktArrival()

This function should implement the actions taken when by a protocol when a packet "arrives" at the node. The packet can have one of three "arrival modes":

- Network: arrival over the network.
- Generated: arrival due to the packet being locally generated (e.g., a user-traffic packet or protocol control packet generated by this node)
- Timeout: arrival due to a retransmission timer going off (indicating the packet should be requeued for retransmission).

The pktArrival() function must be prepared to handle user-traffic packets as well as all the different types of protocol packets that the protocol module defines. User-traffic packets will return a getType() value of TYPE_TRAFFIC, and are also represented by the NAPI+ class N_TrafficPkt. User-traffic packets are generated by the Traffic Generator Library, which is part of the NAPI+ environment.

The pktArrival() function must call baseRegisterAck() for every ack that has been received at the final destination.

Ultimately, pktArrival() must enqueue the packet, delete the packet, or hand the packet up to the next-layer protocol. As stated previously, the NAPI+ environment does not currently support multiple protocol layers; thus, handing the packet up to the next protocol layer simply causes the packet to be deleted by the environment.

An example follow:

```
void Node::pktArrival(N_Pkt* pkt, int arrival_mode, N_Link* rx_link) {
```

```
    ...
    /* test for packets that have reached their final */
    /* destination and process them accordingly */
    ...
    /* process packets that should be forwarded */
    switch (arrival_mode) {
        /* treat all arrival modes the same way in Baseline */
        case ARRIVAL_NETWORK:
        case ARRIVAL_GENERATING:
        case ARRIVAL_TIMEOUT:
            /* find out which link is best for new packet */
            if (isAck(pkt)) {
                link = getLinkIndexToNeighbor(getDst(pkt), 0);
            }
        else if (pkt->getType() == TYPE_UPDATE) {
            link = getLinkIndexToNeighbor(getDst(pkt), 0);
        }
        else {
            link = getBestLinkToDst(getDst(pkt));
            /* if no good link exists, drop packet */
            if (link < 0) {
                envDeletePkt(pkt, getDst(pkt));
                return;
            }
        }
        /* enqueue the packet on the given link's queue */
        _queues[link]->enqueue(pkt);
        envKickLink(link);      // must be done after enqueue()
        return;
    }
}
```

In the example above, the N_Node function envKickLink() is called for the link on whose queue the new packet has just been enqueued. In general, envKickLink() must be called whenever the protocol thinks that a link should be awakened ("kicked") to do a packet transmission. If envKickLink() is called and the associated physical link is currently busy transmitting some other packet, no action is taken. If the link is not busy, the environment calls PNode's linkAvailable() function. (An explanation of linkAvailable() will be given later.) In general, the protocol module should call envKickLink() whenever it is conceivable that an idle link should be awakened in order to examine queued packets for an appropriate packet to transmit. It is conceivable that a protocol implementation will attempt to kick several links in response to the arrival of a single packet.

D.2.1.3 linkAvailable()

This function is called by the environment when a link becomes idle at the end of a previous packet transmission or when a link has been idle and is being kicked (via envKickLink()). The linkAvailable() function must determine which packet is to be transmitted next over the given idle link, and then return a pointer to the packet. It may be that no appropriate packet exists; in this case the function should return 0 or NULL. Returning a packet from this function is the only mechanism by which a NAPI+ protocol module can transmit a packet.

An example follows:

```
N_Pkt* Node::linkAvailable(N_Link* nlink) {
    /* get the pointer to the associated q */
    q = _queues[nlink->getIndex()];

    /* step through the packet queue and find the first one */
    /* which satisfies the criterion for dispatch */
    for (q->first(); !q->atEnd(); q->next()) {
        dst = getDst(q->getCurPkt());
        /* v is the raw link delay and enbr is the e(d) */
        /* reported by the neighbor at the other end of the link */
        exp_delay = (*_v)[nlink_index] +
                    _enbr->getElem(nlink_index, dst);
        if (exp_delay < _emax) {
            pkt = q->getCurPkt();
            break;
        }
    }
    return(pkt);
}
```

D.2.1.4 sendAck()

Since acks are protocol dependent, ack packets can be generated only by the protocol. The environment calls the sendAck() function to instruct the protocol module to create and send an ack. This function is called whenever the node receives a packet that should be acked. We concede that this function is not really necessary, and that it would be sufficient for the protocol to examine received packets in pktArrival() and make the decision to send an ack. Nonetheless, we chose to enforce correct behavior by having the environment call this function whenever the isAckable() function is true for any received packet. (The function is Ackable() is defined by the protocol developer and will be discussed later.) Note that for debugging purposes it may be useful to note that sendAck() is called for an ackable received packet before pktArrival() is called for that same packet.

After the ack is created, it must be handed over to the environment by a call to basePktArrival(), a function of the base class N_Node. In basePktArrival(), the environment does some maintenance processing and hands the packet back to the protocol module via pktArrival() with arrival mode GENERATED. Note that pktArrival() is also implemented in protocol node class, as described earlier.

An example follows:

```
void Node::sendAck(N_Pkt* pkt, int dst) {
    /* create ack packet based on information from the arrived pkt */
    AckPkt* ack = new AckPkt(...);
    /* call basePktArrival to allow the ack pkt to be  enqueued */
    basePktArrival(ack, ARRIVAL_GENERATING, NULL);
}
```

D.2.1.5 getUpdate() and rcvUpdate()

The getUpdate() function must retrieve the value of a protocol variable from the protocol module, and is called at various times when a field in a queued update packet needs refreshing. In general, the protocol developer should keep protocol variables in the PNode class; but this is a design decision, and the developer may choose to put protocol variables almost anywhere.

The function rcvUpdate() must define what to do with a variable that has been received in an update packet from another node. In general, rcvUpdate() should be implemented to set or modify the value of a protocol variable in the receiving node.

Note that multiple instances of getUpdate() and rcvUpdate() may need to be implemented for update variables belonging to matrices, arrays, and single elements. Each type of variable passed in an update packet will require getUpdate() and rcvUpdate() with the appropriate argument set (function prototype). Examine the header file n_node.h in the NAPI+ environment release for the types of getUpdate() and rcvUpdate() functions that must be provided.

Examples of getUpdate() and rcvUpdate():

```
int Node::getUpdate(int var_id, int index) {
    float eval = computeE(index);
    int var = (int) (eval * SCALE_E);
    return var;
}

void Node::rcvUpdate(int reporting_node, float report_time,
                     int var_id, int index, int value) {
    /* Store new update information in appropriate arrays */
    /* In this protocol, enbr(1,d) represents the value e(d) */
    /* received from the nbr at the other end of link 1 */
    for (int l = 0; l < getNumOutLinks(); l++) {
        N_Link* link = getOutLink(l);
        if (link->getNbr() == reporting_node) {
            _enbr_time->set(l, index, time);     // record report time
            _enbr->set(l, index, value);         // record value
        }
    }
}
```

D.2.1.6 wrapupSim()

The wrapupSim() function is called by the environment module when the simulation is finished. The function is not required to perform any actions; however, it is called to give notification to the protocol module that the simulation is finished. The protocol module may elect to do final statistical computations for printing or logging in the simulation log file. Final computations could include such measures as throughput and average end-to-end delays.

D.2.2 Protocol Link Class

The protocol module must define and implement a protocol link class that is derived from the NAPI+ class N_Link. We will refer to this derived link class as PLink. In general, no mandatory functions are required to be implemented for PLink, and the Plink class may in fact be empty. However, certain functions may need to be defined, depending on other factors. For example, if the

protocol module has indicated that it will be sending probe packets, the PLink class must define and implement the function makeProbe(). The following is an example of a definition for the protocol link class PLink:

```
class PLink : public N_Link {
public:
    PLink(N_Node*, int nbr, int link_index, int direction,
         float capacity, float prop_delay, float initProb,
         float initTimeout, int use_default_prob, int link_type,
         float probe_interval);
    virtual ~PLink();

    N_Pkt* makeProbe();
};
```

Some functions will need to be implemented if the argument use_default_prob of the N_Link constructor call is set to 0 (zero). Setting this argument to 0 indicates that the NAPI+ default link probability estimator should not be invoked by the environment. The default link probability estimator is the algorithm defined in the STIP1 report. This algorithm is also used in the Baseline and STIP2 protocols. If the protocol sets the use_default_prob argument to zero (indicating non-use of the algorithm), the protocol developer must provide the following functions in the PLink class:

- registerXmt()
- registerAck()
- recomputeProb().

These functions are discussed in more detail below.

D.2.2.1 makeProbe()

The protocol module indicates that probe packets should be scheduled by setting the probe_interval argument of the N_Link constructor to some nonzero value. The environment, in turn, schedules probe packets to be sent according to the value of probe_interval. When each probe generation time arrives, the environment calls PLink's makeProbe() function.

The following is an example implementation:

```
N_Pkt* PLink::makeProbe() {
    return (new ProbePkt(_node, getNbr()));
}
```

D.2.2.2 registerXmt()

If the default link probability estimator is not being used, the registerXmt() function is called by the environment at the beginning of every packet transmission. If packet transmissions are considered notable by the alternative link probability estimator, this function should be implemented to record the packet transmission for future link probability computation.

D.2.2.3 registerAck()

If the default link probability estimator is not being used, the registerAck() function is called by the environment in response to a call from the protocol module to baseRegisterAck(). Note that the protocol module must call baseRegisterAck() for every received packet that is an ack. In

baseRegisterAck(), the environment performs some initial maintenance processing and subsequently calls registerAck if the default link probability estimator is not being used. While no action is required to be performed by registerAck(), it is expected that this function will be written to record the reception of ack packet information as it pertains to the link probability computation and the link timeout window.

D.2.2.4 recomputeProb()

If the default link probability estimator is not being used, the recomputeProb() function is called at every environment epoch interval (indicated by the input parameter env_epoch_interval). The protocol developer may decide to execute link probability computations in this function, but these computations are not required.

D.2.2.5 timeoutPkt()

If the default link probability estimator is not being used, the timeoutPkt() function is called by the environment whenever the retransmission timer for a packet goes off. While this function is not required to do anything, we expect that the protocol module will implement it to affect the protocol's link probability estimation, when and if appropriate.

D.2.3 Protocol Packet Classes

A protocol packet class must be defined and implemented for every packet type defined by the protocol module (except user-traffic packets). Each packet class must be derived from the NAPI+ base class N_Pkt. Required functions for each packet class include the following:

- getLenBits()
- isAckable()
- getClassName()
- copy()
- A "copy-constructor."

All packets are protocol dependent except for user-traffic packets. These protocol-dependent packets are generally considered control packets. Examples of types of control packets include update or hello packets, acks, and probe packets. Data structures introduced in these control packets are known only to the protocol, and not to the environment. For this reason, the protocol module must define and implement the five functions above for each packet class.

Following is an example of the definition of an update packet, as taken from the Baseline protocol:

```
class UpdatePkt : public N_Pkt {
public:
    UpdatePkt(N_Node* creator, int dst);
    UpdatePkt(const UpdatePkt&);
    ~UpdatePkt();

    /* required functions for N_Pkt derived class */
    int getLenBits() const;
    int isAckable() const { return True; }
    const char* getClassName() const { return ("UPDATE "); }
    N_Pkt* copy() { return (new UpdatePkt(*this)); }
```

```
    /* public functions offered by the derived class */
    int getSrc()            { return _src; }
    int getDst()            { return _dst; }
    float getUpdateTime()   { return _update_time; }
    N_VBList* getData()     { return _vblist; }
    void print(char* indent = "");

    /* private functions within the derived class */
private:
    friend class Node;
    void setDst(int dst) { _dst = dst; }
    void setData(N_VBList* v) { _vblist = v; }

    /* private data content of update pkt, known only */
    /* to the update pkt object */

private:
    const int _src;
    int _dst;
    /* update packet's data content, list of variables */
    N_VBList* _vblist;
};
```

D.2.3.1 getLenBits()

The getLenBits() function must return the length of the packet (in bits) as it would appear "on the air" (on the media). This function is used by the protocol in computing the transmit time (transmit delay) of the packet.

Note that the protocol developer has complete flexibility in adding fields to the protocol's packet classes. These include fields that might be information tags on the packet while the packet is stored at a particular node. The protocol developer must be careful to account for all "on the air" packet fields in the calculation of packet length. Such fields are defined to be those fields that would actually be inserted into a packet's bit format on the link media. At the same time, the packet length must to exclude those fields that are just tags for local-node consumption.

D.2.3.2 isAckable()

This function must be defined and implemented to return an indication of whether or not this packet is "ackable." For a given packet type, the isAckable() function should return 1 (one) if and only if an ack packet should be generated by a node that receives such a packet. The NAPI+ environment calls isAckable() on various occasions:

- At the transmitting node, to determine whether or not to record information in the link-probability estimation data structures
- At the transmitting node, to determine whether or not to record information in the re/ transmission data structures
- At the receiving node, to determine whether or not to call PNode::sendAck().

D.2.3.3 copy() and the Copy Constructor

The copy() function is called by the environment when a copy of the packet is needed. For example, when the environment transmits a packet, the original packet is kept in the node on a retransmission list, and the environment actually sends a copy.

The copy() function must be defined and implemented in each packet class definition as shown below, where "XXXPkt" represents a specific packet class type:

```
N_Pkt* copy() { return new XXXPkt(*this); }
```

Note that copy() makes a call to the copy constructor function. The copy constructor function must make an exact copy of the packet.

D.2.3.4 getClassName()

This function must return a unique name string for each different packet type implemented in the protocol module. Examples of such strings are "UPDATE" and "ACK."

D.2.4 Protocol Timer Class

The Protocol timer class is the mechanism by which the protocol module schedules events to occur in the future. For example, the protocol will undoubtedly want to periodically send some sort of update or hello packet. To enable the protocol to do so, the developer will need access to the protocol timer mechanism, in order to schedule the generation time of each update packet.

The basis for scheduling future events is the implementation of a protocol timer class. All protocol timer classes implemented by the developer must be derived from the NAPI+ base class N_Timer. As part of the class definition, the developer must implement the function callBack() to contain the code that will be executed when the timer expires. To start the timer, the base class N_Timer function set() is called.

There are a number of ways a protocol timer may be implemented. A timer may be of the continuous type. In this case, the base class function set() should be called at the end of the callBack() function. Or, the timer may be a "one-shot" timer. In this case, set() would not be called more than once.

The following example illustrates the class definition and the associated callBack() function of the update timer class of the Baseline protocol:

```
class UpdateTimer : public N_Timer {
public:
    UpdateTimer(PNode* node);
    virtual ~UpdateTimer();

    /* required function */
    void callBack(void* data);

    /* private data */
private:
    float _interval;
    Node* _node;
};

void UpdateTimer::callBack(void* data) {
    /* update timer expires, send update */
    _node->generateUpdates();

    /* set new time out */
    set(_interval, NULL);
}
```

D.3 MISCELLANEOUS MANDATORY FUNCTIONS

The protocol developer must also define several other functions required for the NAPI+ class N_Env. The environment defines the N_Env class and its functions in the file n_env.h, but the protocol must implement some of the functions. These functions include initProtocol(), initLink(), and getQPriorityTraffic().

D.3.1 initProtocol()

This function is called by the environment when a node is to be created. The protocol developer must implement initProtocol() to instantiate (allocate) a new object of the node class (PNode) that is derived from N_Node. The following is an example of initProtocol():

```
N_Node* N_Env::initProtocol(int nodeNum, int numNodes) {
    return (new PNode(nodeNum, numNodes));
}
```

D.3.2 initLink()

This function is called by the environment when a link is to be created. The protocol developer must implement initLink() to instantiate a new object of the link class (PLink) that is derived from N_Link. The following is an example of initLink().

```
N_Link* N_Env::initLink(N_Node* node, int link_index, int nbr,
                        int direction, float capacity_bps,
                        float prop_delay, int link_type)
{
    Link* link;
    if (direction == OUTGOING) {
        link = new PLink(node, nbr, link_index, direction,
                         capacity_bps, prop_delay,
                         1.0, 1.0, 1, link_type,
                         EF_Get_Parm_Float("bl_probe_interval"));
    }
    else {
        /* Incoming links don't set initProb, initTimeout.*/
        /* Incoming links don't set use_default_prob. */
        /* Incoming links don't send probes          */
        link = new Link(node, nbr, link_index, direction,
                        capacity_bps, prop_delay,
                        -1.0, -1.0, 0, link_type, 0.0);
    }
    return (link);
}
```

D.3.3 getQPriorityTraffic()

This function is called by the environment in order to query the protocol module for the queueing priority of user-traffic packets (class N_TrafficPkt), when the protocol module uses the NAPI+ queue class N_PriorityTimeQueue. The N_PriorityTimeQueue class enqueues packets by queueing priority first, and then by arrival time. Note that the protocol module is not required to use the N_PriorityTimeQueue class. If the protocol developer chooses not to use this class, the value returned by getQPriorityTraffic() is ignored. An example of getQPriorityTraffic() follows:

```
#define Q_PRIORITY_TRAFFIC 5
int N_Env::getQPriorityTraffic() {
    return Q_PRIORITY_TRAFFIC;
}
```

Note that if the protocol developer decides not to use the queueing scheme of the N_PriorityTimeQueue class, the developer will have to implement another packet-queue class that must be derived from the class N_PktQueue. In general, the new packet-queue class will consist chiefly of a new enqueue() function.

D.4 MANIPULATING UPDATE DATA IN NAPI+

NAPI+ provides a collection of classes that the protocol module can use to store the values of update variables. The term "update variables" is used here to mean variables that are passed in update or hello-type packets from one node to another. NAPI+ provides several types of classes, so that the protocol can represent update variables as arrays and matrices in different formats. For the purposes of discussion, we will refer to objects of these classes as "variable blocks." Included in these classes are several functions that can be used to manipulate the variable data stored in an object of the associated class.

All variable block classes are derived from the base class N_VB. The specific variable block class that the protocol uses depends on whether the variable represented is a matrix, an array, or a single element. In addition, for matrices and arrays, different classes are provided: the choice of class depends on whether or not the protocol is sending an entire image of the variable (the whole or a part). The following is a list of the variable block classes:

- N_WholeMatrix: filled with the entire image of a matrix variable
- N_WholeArray: filled with the entire image of an array variable
- N_PartArray: filled an indexed set of elements (a "partial array") from an array variable
- N_WholeArrayMatrix: filled with one complete row or column from a matrix variable
- N_PartArrayMatrix: filled with a "partial array" from a row or column of a matrix variable
- N_ElemMatrix: filled with a single element from a matrix variable
- N_ElemArray: filled with a single element from an array variable
- N_ElemBasic: filled with a single element variable.

Once the variable block is created, it can be appended to a list of variable blocks. The NAPI+ class N_VBList can be used by the developer to create such a list. This variable list can then in turn be used as a data element in an update packet (see the example of the class definition of UpdatePkt above). Below is an example of creating a variable block and appending it to a variable block list. The example uses a variable block class N_WholeArray, fills it in with current update data, and appends it to an update packet's variable block list.

```
/* make an array of size getNumNodes() */
N_WholeArray* a = new N_WholeArray(VAR_E, WIDTH_E, getNumNodes());
for (int d = 0; d < getNumNodes(); d++) {
    if (_nodeNum == d) {
        var = 0;
    }
    else {
        /* get current update data for each destination */
        var = getUpdate(VAR_E, d); }
        /* and put it in the whole array object */
    }
    a->set(d, var);
}
/* append variable block a to the variable block list */
list->append(a);
```

The N_Node class provides functions that are useful in manipulating the variable blocks in N_VBList objects. These functions include

- bringCurrent(): refreshes the value fields in all variable blocks in the list.
- reduceOld(): deletes any fields in the variable blocks that do not match the current value of the variables represented.
- combineUpdates(): merges two lists into one list, merging two separate variable blocks into one where appropriate.

D.5 CREATING AND SENDING PACKETS

After a packet is created by the protocol, the N_Node class function basePktArrival() must be called to pass the packet to the environment. After some initial processing, this call generates a call to the protocol's node class pktArrival() function. As explained earlier, pktArrival() is then responsible for enqueueing the packet. An example of generating and sending an update packet is given below:

```
/* create update pkt */
UpdatePkt* upkt = new UpdatePkt(this, ...);

 ..
 * insert variable blocks into the packet */
   ...
/* set the destination of the packet */
upkt->setDst(d);
upkt->setEnvDst(d);

/* call basePktArriavl to hand the packet to the environment */
basePktArrival(upkt, ARRIVAL_GENERATING, NULL);
```

D.6 MANIPULATING PACKET QUEUES IN NAPI+

NAPI+ provides the packet-queue classes N_PktQueue and N_PriorityTimeQueue (which is derived from N_PktQueue). N_PktQueue is simply a linked-list class that can be used to store packets. N_PktQueue's enqueue() function simply adds the given packet to the end of the list. As noted before, N_PriorityTimeQueue's enqueue() function enqueues the given packet by priority and arrival time. The protocol developer can choose to use either of these two classes, or define a new class derived from N_PktQueue.

The NAPI+ implementations of Baseline, STIP2 and STIP3 all use the N_PriorityTimeQueue class for their packet queues. The following shows how the Baseline implementation builds the queue structures. Note that the Baseline protocol has a packet queue for each outgoing link. Also note that this code is implemented in the PNode class start() function.

```
/* _queues is a data element in the protocol PNode class */
_queues = new N_PriorityTimeQueue*[getNumOutLinks()];
for (int i = 0; i < getNumOutLinks(); i++) {
    _queues[i] = new N_PriorityTimeQueue();
}
```

The STIP2 and STIP3 implementations build their queues similarly, except that a queue is built for each destination.

With the exception of the function enqueue(), the same basic queue access functions are used, regardless of whether the developer chooses to use N_PktQueue or a class derived from N_PktQueue. These queue access functions include

- first(): sets the internal pointer (the iterator) to the first packet on the list, and return a pointer to that packet.
- next(): advances the iterator to the next packet in the list and return a pointer to that packet.
- prev(): advances the iterator to the previous packet in the list and return a pointer to that packet.

- last(): sets the iterator pointer to the last packet on the list, and return a pointer to that packet.
- getCurPkt(): gets a pointer to the packet pointed to by the iterator.
- atEnd(): determines whether or not the packet.
- getSize(): gets the number of packets on the queue.

The following code shows an example of how to traverse a queue (from the first element to the last element) in order to examine or otherwise act on each packet:

```
/* tranverse the queue N_PktQueue q */
for (q->first(); !q->atEnd(); q->next()) {
    /* get the packet associated with the current queue element */
    pkt = q->getCurPkt();
    ...
    /* do something with the packet */
    ...
}
```

D.7 NAPI+ ARRAY AND MATRIX CLASSES

NAPI+ provides a collection of classes that the protocol developer can use to store the values of array or matrix data. The chief advantage of using these classes is that they conveniently provide services such as memory allocation, memory deallocation, and index range checking, by simply constructing an object of the class. These classes are

- N_ArrayInt
- N_ArrayFloat
- N_MatrixInt
- N_MatrixFloat.

Below are examples of the use of some of these classes by NAPI+ Baseline protocol.

```
/* _e is an N_ArrayFloat with dimension number of nodes */
_e = new N_ArrayFloat(getNumNodes(), "e", 1);

/* set dth element of the _e array to 0.0 */
(*_e)[d] = 0.0;

/* _enbr is an N_MatrixFloat with dimensions number */
/* of outgoing links and number of nodes.           */
_enbr = new N_MatrixFloat(getNumOutLinks(),getNumNodes(), "enbr", 0);

/* set the [1,d]th element of the _enbr matrix to 0.0 */
_enbr->getElem(1,d) = 0.0;
```

D.8 LOGGING VARIABLE DATA

The NAPI+ environment provides a method for the periodic logging of variable values. To set up a matrix or an array such that the values will be logged, the developer must do the following:

- Instantiate an N_ArrayInt, N_ArrayFloat, N_MatrixInt, or N_MatrixFloat object and keep the values of the variable in the object.

- Set the logFlag argument of the construction of the object to 1 (one). Note that the example for N_ArrayFloat above has the logFlag variable set to 1, indicating that the variable "e" will be logged, but that the variable enbr will not.
- The variable object must be appended to the node object's variable table. The variable table is implemented by the environment as the data member "_var_table" in the base class N_Node, and thus the variable table is automatically included in the protocol's node class (PNode). There is no limit to the number of variables that the developer may append to the variable table.

The following is an example of how to create an array object for storing "e" and how to tell the environment to log the values periodically:

```
/* The last argument of N_ArrayFloat constructor is set to 1 */
_e = new N_ArrayFloat(getNumNodes(), "e", 1);
/* _e is appended to the node's _var_table */
_var_table->append(_e);
```

In reality, one more step which be accomplished to turn on or off the logging of a variable at run time. The rules above dictate how to setup a variable so that logging can be enabled at simulation run-time. To actually enable the logging of the variable, the "log_XXX" parameter in the simulation parameter file (.ef file) must be set to 1 (one) for logging and 0 (zero) for no logging, where XXX is the name of the variable specified in the constructor for the variable object. For example, the following statements would enable logging for e and disable logging for v:

```
log_e:  1
log_v:  0
```

D.9 INPUT PARAMETER VALUES

The NAPI+ environment provides a method for retrieving the value of parameters that are specified in the simulation parameter file (.ef file). The following functions retrieve different types of values:

- EF_Get_Parm_Float(): retrieves a floating point number
- EF_Get_Parm_Int(): retrieves an integer number
- EF_Get_Parm_String(): retrieves a character string.

These functions will cause the simulation to abort if the named parameter is not present in the parameter file. The following functions will not abort if the parameter is not found; rather, they will return a flag indicating that the parameter was missing:

- EF_Get_Parm_Float_Quiet()
- EF_Get_Parm_Int_Quiet()
- EF_Get_Parm_String_Quiet().

The Quiet versions are appropriate for the input parameters used in debugging by the protocol developer. It may not be appropriate for a general user of the simulation to have to worry about parameters that are only used for debugging. The following code shows examples for both types of functions, quiet and forced (nonquiet).

```
/* retrieving bl_emax value from .ef file */
_emax = EF_Get_Parm_Float("bl_emax");

/* retrieving bl_use_forced_acks from .ef file */
int result = EF_Get_Parm_Int_Quiet("bl_use_forced_acks",
                                   &_use_forced_acks);
if (result != EF_SUCCESS) {
    _use_forced_acks = 0;
}
```

In the above code, the environment causes the simulation to abort (in EF_Get_Parm_Float) if emax is not set in the parameter file. However, if bl_use_forced_acks is not set, the EF_Get_Parm_Int_Quiet() returns the value EF_FAILURE; we then see that _use_forced_acks would be set specifically to 0 (off).

D.10 PROTOCOL AND ENVIRONMENT INTERACTION

The following is an example of how various classes and functions might interrelate in response to a node's receipt of a packet (refer to Figure D.2).

In Step 1, a packet (A) is received by a node and the environment passes the packet to the N_Node() class function basePktArrival(). Note that when the protocol generates a new packet, the protocol must pass the packet to basePktArrival().

N_Node's basePktArrival() performs some initial processing and, in Step 2, passes the packet to the pktArrival() function of the protocol's derived node class (PNode).

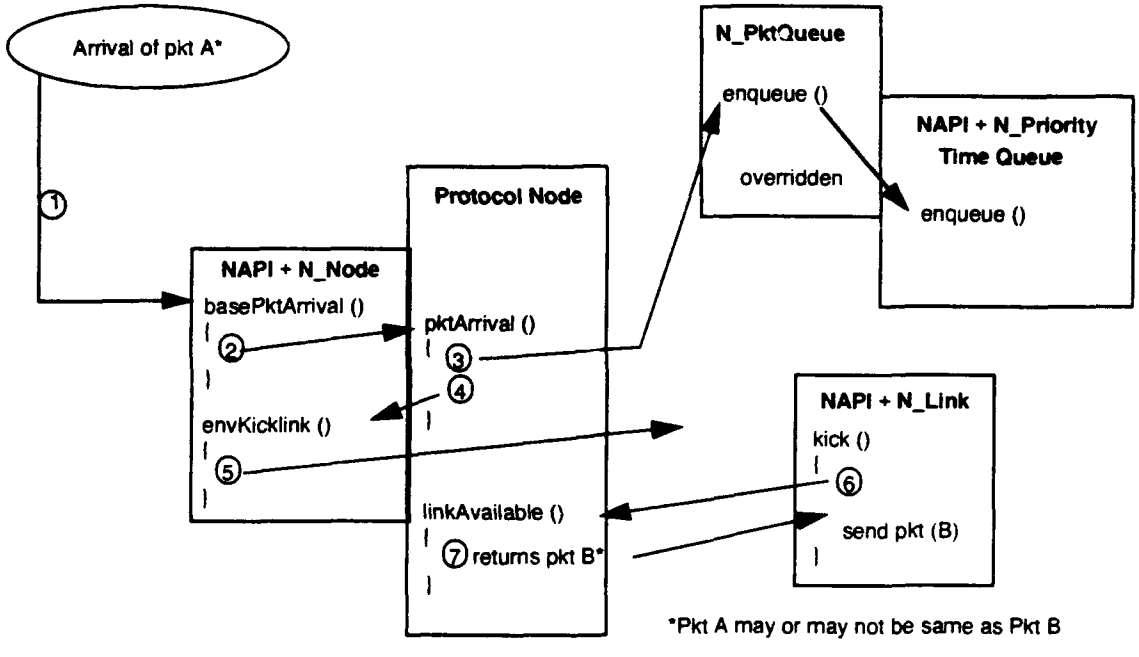


Figure D.2. Class Interaction for Packet Arrival

In this particular example, the protocol determines that the packet should be queued (rather than deleted), chooses the appropriate queue, and calls the enqueue function of that queue (Step 3). Since the selected queue is an N_PriorityTimeQueue, the packet is enqueued by packet priority and time.

The protocol node's pktArrival() function then proceeds to kick an appropriate link (Step 4) by calling envKickLink(). If the protocol uses link queues, then it would be appropriate to kick the link associated with the queue onto which the packet was just put. If the protocol uses a queue per destination, then it may be appropriate to kick each link of the node.

N_Node's envKickLink() function does nothing if the given link is busy. In this case, the link is idle and N_Link's kick() function is called (Step 5).

N_Link's kick() function calls the N_Node's linkAvailable() function to tell the protocol that the link should be awakened and that the protocol should search for a packet to send. By means of the link-scheduling algorithm of the protocol, packet B is found and returned by linkAvailable() (Step 7). Note here that the protocol could also have decided that no packet was eligible to be sent, and could then have returned NULL.

Finally, linkAvailable() returns to kick(), and kick() sends the returned packet.

Here it is crucial to understand that, depending on the protocol, packet B may not be the same packet (packet A) that arrived in Step 1. For example, suppose that destination-oriented queues are used, packet A has the same destination as packet B, and packet A has a higher queueing priority than packet B. Furthermore, suppose that packet B is already queued but the best link on which to send packet B is busy. When packet A is queued, packet B is pushed further back in the queue, perhaps beyond a threshold where it might be eligible to be sent out on a different link. If the protocol decided to kick all the links (as in Step 4), then packet B is sent out on this alternate link (if the alternate link is idle). Note also that in this case packet A would still be queued, since it is now waiting for the busy link for which B was waiting when packet A arrived.

D.11 COMPILING A NAPI+ SIMULATION

In general, most NAPI+ simulation code should be written in C++. C code can be used in the simulation, but cannot be used throughout the protocol module, since it is difficult to access the NAPI+ C++ classes from within C code. Additionally, the simulation should be compiled and linked using the G++ compiler. In developing the NAPI+ environment, SRI used G++ version 2.3.3. While we have no reason believe otherwise, we do not know how using other versions of G++ will affect a protocol simulation.

D.11.1 Other Libraries Used by the NAPI+ Environment

The main part of the NAPI+ environment module is contained in the file libenv.a. In addition, two other SRI-supplied libraries are necessary: libsimgen+.a and libsss+.a. The simgen library is used to read in network topology and link-dynamics information from the simulation's script file (.scr file). The sss library is used to generate random numbers for the simulation.

In addition, the standard UNIX math library should also be linked in using the -lm switch in the link command.

D.11.2 Makefile Miscellaneous

The developer should first ensure that the $EDY environment variable is set. This can be done in the developer's .login file. For example

```
setenv EDY /usr/projectb/edmunds
```

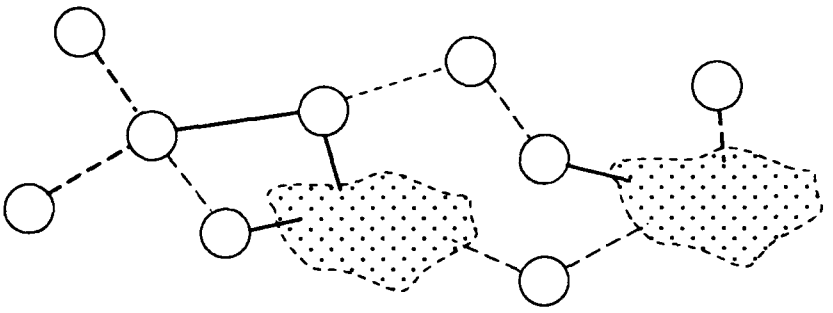
All necessary NAPI+ include files are located in $EDY/sim/include/napi+, and all necessary libraries are found in $EDY/sim/lib. The following sections from a Makefile also assume that the Gnu C++ compiler (G++) is found in the directory $EDY/bin. We have installed G++ in this location on the mohawk workstation at Rome Laboratory. The following is an example Makefile for compiling a protocol module and linking the protocol module with the environment module to form a NAPI+ simulation.

```
IFLAGS  = -I. -I$EDY/sim/include
CFLAGS  = -O $(IFLAGS)
CC      = $(EDY)/bin/g++
GCC     = $(EDY)/bin/gcc
OBJS    = protocol_node.o \
            protocol_link.o \
            protocol_pkt.o \
            protocol_timer.o \
            n_env.o
SRCS    = *.cc
LIBS    = $EDY/sim/lib/libenv.a \
            $EDY/sim/lib/libsimgen+.a \
            $EDY/sim/lib/libsss+.a \
            -lm
PROGRAM = my_protocol

$(PROGRAM): $(OBJS) $(LIBS) $(CC) -o $(PROGRAM) $(OBJS) $(LIBS)

clean:;     /bin/rm -f *.o *.a
depend:;    $(GCC) -M $(IFLAGS) $(SRCS) > Makefile.depend

.SUFFIXES: .o .c .cc
.cc.o:;     $(CC) -c $(CFLAGS) $<
.c.o:;      $(GCC) -c $(CFLAGS) $<

# DO NOT DELETE
include Makefile.depend
```

Appendix E

SECURE TACTICAL INTERNET PROTOCOL 3 (STIP3)

Secure Tactical Internet Protocol 3 (STIP3) [1]



Richard Ogier Diane Lee
SRI International
Menlo Park, California

October 25, 1993

[1]This work was supported by the Defense Advanced Research Projects Agency (DARPA) and Rome Laboratory (RL) under contract number F30602-90-C-0003.

Contents

1 Introduction

SRI International (SRI) is pleased to submit this report, which describes the Secure Tactical Internet Protocol 3 (STIP3), the third in a series of three increasingly efficient and robust protocols developed for the EDMUNDS[1] project. STIP3 represents an exciting new direction for survivable routing/flow control protocols for the tactical environment. Specifically, STIP3 has been developed upon a theoretically sound platform to provide high-throughput, low-delay, fair service in multimedia networks that are subject to jamming. The protocol makes use of alternate routing to select outgoing links (among all media) based on local dynamic link-state and distributed congestion information.

A summary of STIP1-3 and the model on which they are based are given in Section 5 of the EDMUNDS Final Report, to which this report is appended.

1.1 Document Overview

This document details all aspects of the STIP3 protocol, including protocol variables and input parameters, link- and network-level algorithms, and packet processing. The document also provides performance results of simulations run on the protocol.

Each of the STIP3 algorithms is described first in English, then in pseudocode. The main protocol variables, often used by more than one of the algorithms, are defined in Section 2. Each algorithm description includes a list of the main variables that it writes under the heading "Export Variables"; and a list of the main variables that it reads under the heading "Import Variables." Variables used only within the algorithm are listed and defined under the heading "Local Variables."

Within the pseudocode, boldface type is used for function names and keywords (such as **for all** (...) **do** {...}, **if** (...) **do** {...}, and **return** (...)). The left arrow (←) is used for assignment. Italics are used for comments that may appear immediately after the function name and above the function body, within the function body in parenthesis, or in the right margin.

[1] EDMUNDS: Evaluation and Development of Multimedia Networks in Dynamic Stress.

2 Variables, Parameters, and Functions

2.1 Primary Variables

Many of the sections in this document include a list of variables that are computed by an algorithm in another section and are "imported" into the section as global variables. (These variables may or may not actually have global scope in a coded implemenation.) In addition, these sections also include a list of "exported" variables to be used by other sections. The following lists all of the export and import variables and a few other key variables of the protocol. The variables are categorized by the nodes and/or links to which they pertain.

- Variables and constants pertaining to links are as follows:
  - $p(l)$ is the current time-smoothed probability of success across link l.
  - $c(l)$ is the capacity of link l in bits/s.
  - $u(l)$ is the propagation delay of link l in s.
  - $c'(l)$ is the effective capacity of link l set equal to $c(l)p(l)$.
  - $v(l)$ is computed to be the raw delay of link l. This delay accounts for the propagation, transmission, and retransmission delays for link l.
  - $w(l)$ is the time-varying timeout period for retransmissions.
  - $w_o(l)$ is the initial timeout period for retransmissions.
  - $cntrl\_flow(l)$ is the time-smoothed measured rate (in bits/sec) of control traffic on link l.
  - $cntrl\_count(d)$ counts the control traffic for computing $cntrl\_flow(l)$.
  - $T(l)$ is the period over which link l examines past packet transmissions to update the link probability.
- Variables pertaining to link-destination pairs are as follows:

- $D(l,d)$ is the delay to destination d over link l set equal to the raw delay over link l plus the neighbor's expected delay $(v(l) + e(j,d))$.
- $f(l,d)$ is the optimal effective flow (in bits/sec) on link l for packets with destination d.
- $g(l,d)$ is the offered flow on link l for packets with destination d. This is equal to $f(l,d)/p(l)$.
- $\Theta(l,d)$ (bits) determines the highest packet in the queue for destination d that is eligible to be transmitted over link l. $\Theta(l,d)$ is always 0 for the best ("primary") link l to destination d.
- $W(l,d)$, $U(l,d)$, and $G(l,k)$ are variables used for link scheduling.

- The variable pertaining to neighbor-destination pairs, $e\_nbr(j,d)$, is the most recently received $e(d)$ value from node j.

- Variables pertaining to destinations
  - $e(d)$ is the official expected delay from a node to destination d. It is sent to neighbors whenever it changes.
  - $e\_priv(d)$ is the private smoothed expected delay that is used for updating $e(d)$.
  - $DAG(d)$ is the set of outgoing links in the current DAG for this node. These are the links over which a positive flow to destination d can currently be assigned.
  - $SDAG(d)$ is a subset of $DAG(d)$ consisting of links l such that $f(l,d) > 0$ or $primary(d) = l$. $SDAG(j,d)$ is equal to 1 if links to j are in $SDAG(d)$ and 0 otherwise. $SDAG(j,d)$ is reported to neighbor j when it is updated.
  - $primary(d)$ is the best link for destination d. Only links in $DAG(d)$ can be the primary link. During transitory states caused by dynamic situations, it is possible that $DAG(d)$ will contain no links. In this case, d has no primary link and $primary(d) = -1$.
  - $q(d)$ is the current queue size in bits for packets with destination d.
  - $\bar{q}(d)$ is the time-smoothed queue size for packets with destination d.
  - $x(d)$ is the time-smoothed measured arrival rate (in bits/sec) for packets with destination d.
  - $arrival\_count(d)$ counts the arrival traffic for computing $x(d)$.

- Other variables are listed below.
  - *jammed* indicates whether the node appears to be jammed, and is sent to neighbors when it changes.

- Xmt_List is a list shared by all links containing the fields (transmit done time, link, ack flag, source, dest, type, seq. no.) for all packets transmitted by this node. The list is indexed by a Xmt_Number that is unique for each transmission of traffic, probe, and update packets. Transmit done time is the time at which the transmission of the packet is completed at the transmitting node.

2.2 Input Parameters

The following input parameters can be used to tune the behavior of the algorithm.

- Δt, also called the epoch interval, is the period at which much of the protocol's computation is performed. In an actual system, Δt may vary from epoch to epoch.
- *slot_size* is the period at which some simple computations, such as incrementing $tokens(l, d)$, is done. Δt should be a multiple of *slot_size*.
- δ and ϵ determine the minimum required change in $e\_priv(d)$ before $e(d)$ is updated.
- μ is the rate at which the minimum required change in $e\_priv(d)$ decays exponentially to zero.
- σ determines the degree to which traffic is spread over multiple paths.
- *spread_DAG* is a binary parameter that, when 1, encourages the DAGs to react to the need for spreading traffic on multiple paths.
- *indirect_routing* is a binary parameter that, when equal to 1, allows the routing of updates and acks over indirect paths to each neighbor, as opposed to routing only over the direct links.
- *link_bias* is a constant that is added to the propagation delay of each link. A large value encourages routing over min-hop paths.
- M is the exponent for the expected delay in the objective function that is used for computing flows when $old\_flows = 1$.
- W_B, W_M, W_S are used to update the timeout window size $w(l)$, based on received acks.
- W_T is used to compute $T(l)$, which is set to $W_T w(l)$.
- α_{inc} and α_dec are smoothing parameters used to update the expected delay $e\_priv(d)$.

- β is the smoothing parameter used to update the measured arrival rate $x(d)$.
- β' is the smoothing parameter used to update the measured control traffic rate $cntrl\_flow(l)$.
- η is an exponential decay rate used in computing the instantaneous probability $p'(l)$.
- γ is the smoothing parameter used to update the smoothed link probability $p(l)$.
- ϕ is the exponential decay rate for computing the future time-averaged queuing delay.
- *emax* is the maximum value permitted for $e(d)$.
- *step* is the step size for gradient descent.
- *max_step*, multiplied by $c'(l)$, is the maximum amount $f(l,d)$ can change per iteration of gradient descent.
- *num_steps* is the maximum number of steps for gradient descent.
- ρ is such that, if all flows $f(l,d)$ change by less than ρ in an iteration of gradient descent, then convergence is assumed.
- κ is a coefficient used in computing the scheduling variable $W(l,d)$.
- *flow_control* is a binary variable that allows traffic packets originating at a node to be dropped if $e(d)$ is too large.
- *priority_k_rate* and *priority_k_burst*, for $k = 0, 1, 2$, are the leaky-bucket parameters for limiting the precedence of packets of priority k.
- $\bar{P}$ estimates the average packet size in bits, for computing transmission delay.
- *auto_gamma* is a binary parameter that, when equal to 1, allows γ to adapt automatically to the rate of link dynamics.
- *use_old_flows* is a binary parameter that determines whether the old STIP2 method or the new STIP3 method is used for computing flows.
- *use_thetas* is a binary parameter that, when equal to zero, forces $\Theta(l,d) = 0$, so that the Θ's are not used.
- *ack_life*, *update_life*, and *traffic_life* are the age limits (in seconds) of ack, update, and traffic packets.

- *detect_jam* is a binary variable that, if equal to 1, allows a node to detect that it is jammed.
- *short_update_period* is the minimum number of epochs between updates.
- *long_update_period* is the minimum number of epochs between low-priority updates.
- *probe_interval* is the time between probe packet transmissions.
- *use_forced_acks* is a binary parameter such that, when both it and *indirect_routing* are equal to 1, then acks are forced on any idle direct links, in addition to being routed as usual.

2.3 Packet Variables

A number of variables are used to refer to the size of a packet and to the fields within the STIP header of packets. These are listed below for a packet P.

- P_t is the packet type set to one of the following:
  - Ack for update
  - Ack for traffic
  - Ack for probe
  - Update
  - Traffic
  - Probe.
- P_x is the Xmt_Number given to packet P and is used by the transmitting node to uniquely identify a transmission of a traffic, update, or probe packet. For ack packets, P_x is the Xmt_Number of the packet being acknowledged.
- P_s is the source node of the packet.
- P_d is the destination of for the packet.
- P_q is the sequence number generated by the node at which the packet initially arrived or was generated. The triplet $< P_t, P_s, P_q >$ uniquely identifies a particular packet in the network.
- P_{gen} is the generation time of the packet.
- P_a is the time of arrival of a packet at this node. This variable is maintained only at the node (it is not part of the STIP packet header).
- $|P|$ is the size in bits of packet P, including the STIP header.

2.4 Functions

The following is a list of all functions included in the pseudocode sections of this document, along with the page numbers of those sections.

3 Epoch Processing

As frequently as possible, the algorithm performs "epoch processing" to update the link and routing variables of the protocol and to generate update packets to neighbors if appropriate. This is the primary processing loop in the protocol. In an actual multiprocessor packet switch, at least one processor would be dedicated to the job of performing epoch processing continuously. In the napi+ simulation, epoch processing is performed periodically, every Δt seconds.

Epoch Processing() is actually called once per time slot, that is, every *slot_size* seconds, to allow more frequent updating of certain variables. However, the main processing is done every Δt seconds. Typical values for these parameters are *slot_size* = 0.1 and $\Delta t = 0.2$.

3.1 Pseudocode

The pseudocode for epoch processing follows.

Epoch Processing()

***Description:** Epoch processing is a procedure that is run continuously (or as frequently as possible) to update the link and routing variables and to generate update packets if appropriate.*

{

Update Queue Sum() $\rightarrow q\_count(d)$

Update G() $\rightarrow G(l,k)$

$slot \leftarrow slot + 1$ *Increment slot count.*

$slots\_per\_epoch \leftarrow \Delta t / slot\_size$

if (*slot* is divisible by *slots_per_epoch*) **do** {

if (any incoming link is busy) **do** $jammed \leftarrow 0$

Update Link Probabilities() $\rightarrow p(l)$

Update Measurements() $\rightarrow x(d), q(d), cntrl\_flow(l)$

Update DAG() $\rightarrow DAG(l,d)$

Update Link Delays() $\rightarrow D(l,d), primary(d)$

Update Flows() $\rightarrow f(l,d), g(l,d)$

Update Private Distance Variables() $\rightarrow e\_priv(d)$

Update Reported Distance Variables() $\rightarrow e(d)$

Update SDAG() $\rightarrow SDAG(l,d)$

Generate Update Packets()

Update W() $\rightarrow W(l,d)$

Update Queue Thresholds() $\rightarrow \Theta(l,d)$

if ($detect\_jam = 1$) **do** $jammed \leftarrow 1$ *jammed will be set to 0 by next epoch if any packet is received.*

for all (l) **do Kick This Link**(l)

}

}

4 Link Algorithms

4.1 Link Probability

Each link estimates the current smoothed link probability $p(l)$ at each epoch. This section discusses how this computation is performed.

4.1.1 Global Variables

Export variables

$p(l)$

Import variables

Xmt_List	$T(l)$	$w(l)$
γ	Δt	t_{mad}

4.1.2 Algorithm

The algorithm detailed below estimates the current link probability, based on whether or not ack packets are received during a link ack interval (which is a number of times larger than the ack timeout window $w(l)$ used to indicate when non-acked packets should be retransmitted). The link probability $p(l)$ thus attempts to estimate the likelihood that a packet transmitted across link l will result in a corresponding ack being successfully received at some time in the future, as opposed to the probability that no retransmissions will be required. The algorithm is run periodically as part of the epoch processing procedure. This algorithm provides faster response to links coming up than the corresponding algorithm for STIP2. This is achieved by computing two different estimates of $p(l)$, one of which is a lower bound, and then setting $p(l)$ equal to the maximum of these estimates. In addition, an option is provided to allow the smoothing constant for $p(l)$ to be adjusted according to the measured link dynamics.

Local Variables The following variables are used by the algorithm for computing the probability of a particular link l (see Figure 4.1):

- $p'(l)$ is the instantaneous probability of link l.

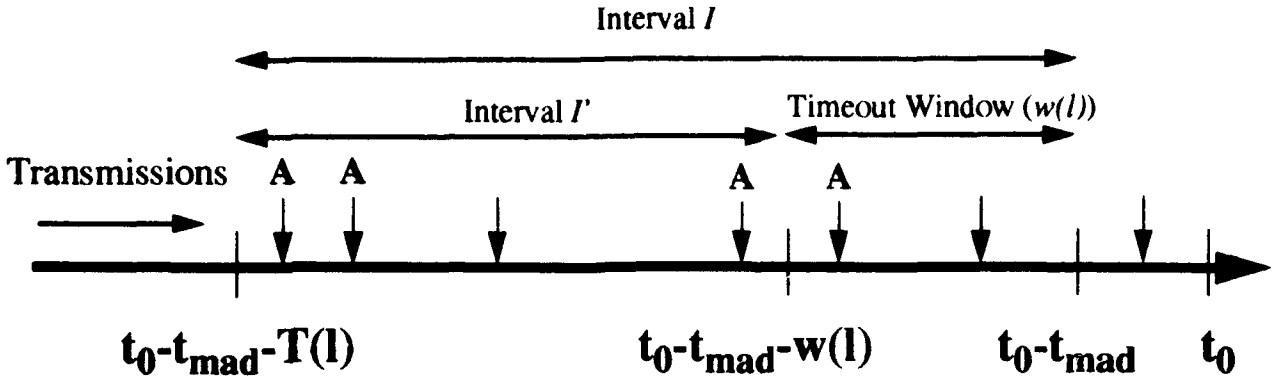


Figure 4.1: Computing Link Probabilities. In this example, seven past packet transmissions for a particular link l are being tracked. Only the four with the interval I' are used to compute $p_1(l)$, and the six within the interval I are used to compute $p_2(l)$. The letter A indicates which transmissions have already been acked.

- $\lambda(l)$ is the measured rate at which $p'(l)$ crosses 0.5.
- $\gamma(l)$ is a dynamic version of γ that is used if the *auto_gamma* option is selected.
- X_k is the time that transmission k over link l occurred. X_k is derived from the above import variable Xmt_List.
- A_k is equal to 1 if transmission k has been acked and 0 otherwise. A_k is also derived from Xmt_List.
- I is the link ack interval of the time from $t - t_{\text{mad}} - T(l)$ to $t - t_{\text{mad}}$. Hence, I is of length $T(l)$.
- I' is the subinterval of I from $t - t_{\text{mad}} - T(l)$ to $t - w(l)$.
- t is the current time.

First, two probability estimates based on different sets of information are computed. The estimate $p_1(l)$ is the probability considering only transmissions that occurred within I', and is given by

$$p_1(l) = \frac{\sum_{k:X_k \in I'} A_k e^{-\eta(t-X_k)}}{\sum_{k:X_k \in I'} e^{-\eta(t-X_k)}} \quad , \tag{4.1}$$

where η is an input parameter that determines the degree to which more weight is placed on the more recent transmissions. The estimate $p_2(l)$ is the probability considering all transmissions that occurred within I, assuming the worst case (that

the transmission will not succeed) for transmissions that have been neither acked nor timed out. It is given by

$$p_2(l) = \frac{\sum_{k:X_k \in I} A_k e^{-\eta(t-X_k)}}{\sum_{k:X_k \in I} e^{-\eta(t-X_k)}} \quad . \tag{4.2}$$

Note that $p_1(l)$ is based on partial information, and that $p_2(l)$ is based on complete information, but provides only a lower bound, since it assumes the worst case for some transmissions. Therefore, it is clear that the maximum of these two estimates provides a good estimate for $p'(l)$, i.e.,

$$p'(l) = \max\{p_1(l), p_2(l)\} \quad . \tag{4.3}$$

The time smoothed probability $p_t(l)$ is computed as follows:

$$p_t(l) = e^{-\gamma\Delta t} p_{t-\Delta t}(l) + (1 - e^{-\gamma\Delta t}) p'(l) \quad . \tag{4.4}$$

The choice for the input parameter γ depends on the assumptions regarding the network's operating environment. Through experimentation, we found that $\gamma = 1$ works well when $p'(l)$ crosses 0.5 more than once per second, and that $\gamma = 100$ works well when $p'(l)$ crosses 0.5 less than once per second.

If the input parameter *auto_gamma* is equal to 1, then for each link l, γ is replaced by $\gamma(l)$, which is initially equal to γ but is updated dynamically as follows. STIP3 computes the time-smoothed rate $\lambda(l)$ at which $p'(l)$ crosses 0.5, and dynamically sets $\gamma(l)$ to 1 if $\lambda(l) > 1$, and to 100 otherwise. The exact procedure is defined by the pseudocode for the function Update Gamma(l), given in the following subsection.

4.1.3 Pseudocode

The following functions detail the code required to implement the link probability computations.

Update Link Probabilities()

Description: *Updates the probabilities $p_t(l)$ for all outgoing links based on received acks. Initially,* $p_0(l) = 1$.

{

for all (l) **do** {

$I \leftarrow$ interval $[t - t_{\text{mad}} - T(l),\ t - t_{\text{mad}}]$

$I' \leftarrow$ interval from $[t - t_{\text{mad}} - T(l),\ t - w(l)]$

From Xmt_List, read X_k and A_k for all transmissions over link l that occurred within interval I.

Compute $p_1(l)$ according to Equation 4.1.

Compute $p_2(l)$ according to Equation 4.2.

$p'_{\text{old}}(l) \leftarrow p'(l)$

$p'(l) \leftarrow \max\{p_1(l), p_2(l)\}$

if (*auto_gamma* $= 0$) **do** {

$p_t(l) \leftarrow e^{-\gamma \Delta t} p_{t-\Delta t}(l) + (1 - e^{-\gamma \Delta t}) p'(l)$

}

else do {

Update Gamma$(l, p'_{\text{old}}(l), p'(l))$

$p_t(l) \leftarrow e^{-\gamma(l) \Delta t} p_{t-\Delta t}(l) + (1 - e^{-\gamma(l) \Delta t}) p'(l)$

}

}

}

Update Gamma($l, p'_{\text{old}}(l), p'(l)$)
Description: Updates $\gamma(l)$ *and the measured rate* $\lambda(l)$ *at which* $p'(l)$ *crosses 0.5. Initially,* $\gamma(l) = \gamma$ *and* $\lambda(l) = 0$.
{

if ([$p'_{\text{old}}(l) < 0.5$ and $p'(l) > 0.5$] or [$p'_{\text{old}}(l) > 0.5$ and $p'(l) < 0.5$]) **do** {

$\lambda(l) \leftarrow (1 - \Delta t/2)\lambda(l) + 0.5$

}

else do {

$\lambda(l) \leftarrow (1 - \Delta t/2)\lambda(l)$

}

if ($\lambda(l) > 1$) **do** $\gamma(l) \leftarrow 1$

else do $\gamma(l) \leftarrow 100$

}

Process Ack Probability(l, **Xmt_Number**)
Description: Sets ACK_RECEIVED for this transmission; this information is used by **Update Link Probabilities**.
{

Ack flag in Xmt_List(Xmt_Number) $\leftarrow$ ACK_RECEIVED

}

4.1.4 Note

- It is important that the length of the interval I' (i.e., $T(l) + t_{\text{mad}} - w(l)$) be large enough so that at least a few transmissions are likely to occur within I'.

4.2 Ack Timeout and Link Ack Windows

Each link adaptively updates the size of the Link Ack Window for use in computing the link probability, and also updates the size of the Ack Timeout Window for use in determining when packets should be retransmitted. The computation of these window sizes is described in this section.

4.2.1 Global Variables

Export variables

$T(l)$ $w(l)$

Import variables

W_B W_M W_S W_T

4.2.2 Algorithm

Local Variables and Constants

- $\bar{d}_A(l)$ is the average time (in seconds) for receiving an ack for transmissions over l, measured from the time that the original packet transmission is completed to the time that the incoming ack is received.
- $\bar{v}_A(l)$ is the variance in delay for receiving acks for transmissions over l.
- $T_{\max}(l)$ is the maximum value allowed for $T(l)$, and is equal to W_M times $w_0(l)$, where W_M is an input parameter and $w_0(l)$ is the initial timeout period. (Our implementation used $w_0(l) = 1$ second and $W_M = 4$.)
- W_V and W_A are constants used to compute $w(l)$.

The algorithm computes, for each outgoing link, a smoothed average of the time required to receive acks for packets transmitted recently. The ack timeout window $w(l)$ is then set to this average, plus W_V times the variance, plus W_A. (Our implementation used $W_V = 2$ and

$$W_A = 1000/c(l) + 5[u(l) + 50/c(l)] \quad , \tag{4.5}$$

which is the transmission delay of a typical (1000-bit) traffic packet, plus five times the sum of the link propagation delay and the transmission delay of a typical (50-bit) ack packet.)

The window size $T(l)$ is then computed to be W_T times $w(l)$, and the minimum ack delay t_{mad} is computed to be W_B times $w(l)$. (Our implementation used $W_T = 3$ and $W_B = 0.2$.) If $T(l)$ exceeds $T_{\max}$, then $T(l)$ is reduced to $T_{\max}$, $w(l)$ is set to $T(l)/W_T$, and t_{mad} is set to $W_B w(l)$.

Initially, $w(l) = w_0(l)$, $T(l) = W_T w_0(l)$, and $t_{\text{mad}} = W_B w_0(l)$. (Our implementation used the initial values $\bar{d}_A(l) = 1$ and $\bar{v}_A(l) = 0$.)

4.2.3 Pseudocode

Process Ack Window(l, Xmt_Number)
Description: Updates the window sizes in response to a received ack.
{

Xmt_Time ← time in Xmt_List(Xmt_Number)
Delay ← t − Xmt_Time *t is the current time*
$\bar{d}_A(l) \leftarrow e^{-W_S \Delta t} \bar{d}_A(l) + (1 - e^{-W_S \Delta t})$ Delay
$\bar{v}_A(l) \leftarrow e^{-W_S \Delta t} \bar{v}_A(l) + (1 - e^{-W_S \Delta t})(\bar{d}_A(l) - \text{Delay})^2$
$w(l) \leftarrow \bar{d}_A(l) + W_V \sqrt{\bar{v}_A(l)} + W_A$
$T(l) \leftarrow W_T w(l)$
if $(T(l) > T_{\max})$ **do** {
$T(l) \leftarrow T_{\max}$
$w(l) \leftarrow T(l)/W_T$
$n_{ad} \leftarrow W_B w(l)$
}

}

4.2.4 Note

- W_A should be set approximately to the the time required to transmit multiple acks plus an average-size traffic packet. This setting would permit the ack to be delayed behind an ongoing transmission and enqueued behind a number of other acks.

4.3 Link Scheduling

Whenever an outgoing link becomes available or is kicked, the link scheduling algorithm decides which enqueued packet, if any, will be transmitted next on that link. STIP1 used a frame-based scheduling algorithm in which each link served the destination queues in weighted round-robin fashion, using the computed flows $g(l, d)$ as the weights. However, this method was found to be unfair to bursty traffic, since each destination was allocated a fixed number of bits per frame. STIP2 used an improved link scheduling algorithm, in which a leaky bucket was maintained for each (l, d) pair, and each link served the destination with the most "credits." Two drawbacks were discovered with this method:

- Traffic of a given destination was not guaranteed to be divided among outgoing links in proportion to the flows.

- Traffic with large bursts was not treated fairly, since the computed flows are based on time-smoothed measurements of the incoming traffic.

The STIP3 link scheduling algorithm overcomes both drawbacks, while preserving the following benefits:

- Destination queues are still used, so that the routing decision can be made at the last possible moment.

- A link l that has higher delay than the primary link for some destination d may select a packet from the middle or tail of queue d, depending on the computed variables $\Theta(l,d)$, in order to minimize maximum packet delay.

In addition, the STIP3 link scheduling algorithm supports multiple priorities, and includes a method that allows the limiting of high priority traffic (if desired) in order to prevent lower priority traffic from being squeezed out.

4.3.1 Global Variables

Export variables

$cntrl\_count(l)$

Import variables

$g(l,d)$	$\Theta(l,d)$	κ
$primary(d)$	$c(l)$	

4.3.2 Algorithm

Local Variables The link scheduling algorithm uses three variable matrices:

- $U(l,d)$ represents the amount of d-traffic (in bits) that has been sent on link l in excess of the amount that would have been sent if d-traffic were divided among the links in exact proportion to the flows $g(l,d)$. (By d-traffic we mean traffic destined for d.)

- $W(l,d)$ represents the amount of traffic in the d-queue that has been assigned to link l.

- $G(l,k)$ represents the amount of available credit for traffic of priority k on link l.

Updating the Local Variables Initially, $U(l,d) = 0$ for all l and d. Every time a d-packet is sent on some link l, $U(l,d)$ is increased by the size $|P|$ of the packet, and

$U(l', d)$ for each l' is decreased by $|P|$ times the fraction of d-flow that is assigned to link l', i.e.,

$$U(l, d) \leftarrow U(l, d) + |P| , \tag{4.6}$$

$$U(l', d) \leftarrow U(l', d) - \frac{g(l', d)}{\sum_{l''} g(l'', d)} |P| , \quad \text{for all } l' . \tag{4.7}$$

This updating rule ensures that the equation

$$\sum_{l} U(l, d) = 0 \tag{4.8}$$

is always maintained. A negative value of $U(l, d)$ implies that more d-traffic needs to be sent on link l to agree with the flows.

It is easy to see that if we always assigned an incoming d-packet to the link l such that $U(l, d)/g(l, d)$ is minimized, we would ensure that d-traffic is divided among the links in proportion to the flows. However, we cannot do this since we do not use link queues: This motivates the need for $W(l, d)$.

We emphasize that the algorithm does not assign individual packets to links, since this would be equivalent to using link queues. Instead, a certain *amount* of d-traffic, represented by $W(l, d)$ is assigned to each link l.

Initially $W(l, d) = 0$ for all l and d. When a d-packet arrives, the link l is chosen that *minimizes*

$$[\kappa W(l, d) + U(l, d) + \omega]/g(l, d) \tag{4.9}$$

for some constants κ and ω; and $W(l, d)$ is increased by the packet size. When a d-packet is sent on a link l, $W(l, d)$ is decreased by the packet size.

If $\kappa = 1$ and $\omega = 0$, then it is easy to see that traffic is assigned to links in proportion to the flows $g(l, d)$. However, we discovered in simulations that using $\kappa = 1$ sometimes caused more packets to be assigned to a link than the link could handle, due to the actual incoming flow being temporarily larger than its time-smoothed average. Using $\kappa = 100$ avoids this problem by emphasizing $W(l, d)$ more. For convenience, κ is an input parameter in our implementation of STIP3. In addition, we chose $\omega = 1$ bit so that the link with the most flow is selected when $W(l, d) = U(l, d) = 0$ for all l.

Selecting the Packet A packet is defined to be *above* $\Theta(l, d)$ in queue d if $n > \Theta(l, d)$, where the first bit of the packet is the nth bit of the queue. In particular, if $\Theta(l, d) = 0$, then the first packet above $\Theta(l, d)$ is the first packet in the queue.

The function **Find Eligible Packet(l)** uses the following procedure to select the next packet, if any, to be transmitted on link l. First, a priority k is selected (discussed below). If the priority is that of a traffic packet, then **Select Traffic Packet(k, l))** is called, which uses the following rules (these rules are stated more formally in the pseudocode):

- Among all d' with $W(l,d') > 0$, the destination d is selected such that the first packet of the given priority on queue d has the earliest arrival time (if such a d exists).
- If $\Theta(l,d) = 0$ (which is the case if *use_theta* $= 0$), the selected packet is the first packet of the given priority on queue d.
- Otherwise, the selected packet is the first packet of the given priority above $\Theta(l,d)$ on queue d, or, if no such packet exists, the last packet of the given priority on queue d.

If $k = 0$, which is the priority of a control packet (i.e., an update or ack), then **Select Control Packet(l)** is called, which uses slightly different rules (described in the pseudocode). In particular, if *indirect_routing* $= 1$, then control packets are treated similarly to traffic packets. However, if *indirect_routing* $= 0$, then control packets are restricted to using the direct link to the destination.

Scheduling Priorities Scheduling priorities are to be distinguished from queuing priorities, which are discussed in Section 5.2. STIP3 implements three scheduling priorities: priority 0 for control packets, and priorities 1 and 2 for traffic packets. STIP3 can easily be modified to handle a larger number of priorities.

A simple way to handle priorities (which is allowed in STIP3) is to first try to find an eligible packet of priority 0, and if that fails, repeat the same process for priorities 1 and 2. This method gives unlimited precedence to high priority traffic, which may be appropriate is some cases. However, in other cases we may wish to restrict the amount of bandwidth that each priority can use, so that the lower priorities are not squeezed out. For example, we may wish to limit the amount of bandwidth used by control packets.

STIP3 accomplishes this restriction by maintaining a leaky bucket for each link-priority pair. Each such leaky bucket is represented by the variable $G(l,k)$, $k = 0,1,2$, which represents the available credit for priority k on link l. The leaky bucket corresponding to (l,k) has two parameters: the rate parameter $\rho_{l,k} =$ *priority_k_rate* $* c(l)$ and the burstiness parameter $\sigma_{l,k} =$ *priority_k_burst* $* c(l)$. At each time slot (see Epoch Processing), **Update G()** is called, which increments $G(l,k)$ at the constant rate $\rho_{l,k}$, but does not allow $G(l,k)$ to exceed $\sigma_{l,k}$. Whenever **Find Eligible Packet(l)** selects a packet of priority k, it reduces $G(l,k)$ by the packet length.

By using leaky buckets, we allow high priority traffic to have precedence as long as it does not use more than the specified amount of bandwidth in the long run, even if it is bursty. For example, if priority_0_rate = 0.2, then control traffic is guaranteed precedence only if it does not use more than 20% of the link capacity.

When **Find Eligible Packet(l)** is called, it first calls **Select Control Packet(l)** or **Select Traffic Packet(k, l)**, as appropriate, for each priority until a packet is found. It first chooses the priorities (from 0 to 2) such that $G(l,k) > 0$. If no packet has been selected, it then chooses the priorities (from 0 to 2) such that $G(l,k) \leq 0$.

4.3.3 Pseudocode

Find Eligible Packet(l)

Description: Returns the next packet to be transmitted on link l, and updates G, U, W, and cntrl_count(l).

{

(Select packet.)

if ($G(l,0) > 0$) **do** $P \leftarrow$ **Select Control Packet**(l)

if (P = NULL and $G(l,1) > 0$) **do** $P \leftarrow$ **Select Traffic Packet**(1,l)

if (P = NULL and $G(l,2) > 0$) **do** $P \leftarrow$ **Select Traffic Packet**(2,l)

if (P = NULL and $G(l,0) \leq 0$) **do** $P \leftarrow$ **Select Control Packet**(l)

if (P = NULL and $G(l,1) \leq 0$) **do** $P \leftarrow$ **Select Traffic Packet**(1,l)

if (P = NULL and $G(l,2) \leq 0$) **do** $P \leftarrow$ **Select Traffic Packet**(2,l)

(If no eligible packet was found, return NULL.)

if (P = NULL) **do return** NULL

(Measure control traffic for adjusting link capacity.)

if (P is a control packet) **do** $cntrl\_count(l) \leftarrow cntrl\_count(l) + |P|$

(Update $G(l,k)$.)

$k \leftarrow$ priority of P

if ($G(l,k) > 0$) **do** $G(l,k) \leftarrow G(l,k) - |P|$

$d \leftarrow$ destination of P

(Update $W(l,d)$.)

if ($f(l,d) > 0$ or [$primary(d) = l$ and
(P is a traffic packet or $indirect\_routing = 1$)]) **do** {

$W(l,d) \leftarrow W(l,d) - |P|$

}

(Update $U(l,d)$.)

if ($g(l,d) > 0$ and [P is a traffic packet or $indirect\_routing = 1$]) **do** {

$U(l,d) \leftarrow U(l,d) + |P|$

for all (l') **do** {

$$U(l',d) \leftarrow U(l',d) - |P| \frac{g(l',d)}{\sum_{l''} g(l'',d)}$$

}

}

Dequeue P

return P

}

Select Control Packet(l)

Description: Finds the next control packet to be transmitted on link l.

{

if (*indirect_routing* = 0) **do** {

$d \leftarrow$ tail node of l

$P \leftarrow$ first packet on queue d

if (P is a control packet) **do return** P

else do return NULL

}

if (*indirect_routing* = 1) **do** {

$A \leftarrow \{d$: $W(l,d) > 0$ or (*primary*$(d) = -1$ and d is the tail node of $l)\}$

$B \leftarrow \{d$: $d \in A$ and queue d has a control packet $\}$

if (B is empty) **do return** NULL

$d \leftarrow$ member of B that minimizes P_a^d,
where P^d is the the first control packet on queue d

$P \leftarrow$ first control packet on queue d

return P

}

}

Select Traffic Packet(k, l)

Description: Finds the next traffic packet of priority k to be transmitted on link l.

{

$B \leftarrow \{d$: $W(l,d) > 0$ and queue d has a packet of priority $k\}$

if (B is empty) **do return** NULL

$d \leftarrow$ member of B that minimizes P_a^d,
where P^d is the the first packet of priority k on queue d

if (there is a packet of priority k above $\Theta(l,d)$ on queue d) **do** {

$P \leftarrow$ first packet of priority k above $\Theta(l,d)$ on queue d

}

else do {

$P \leftarrow$ last packet of priority k on queue d

}

return P

}

Update W for Arrival(P)

Description: Updates $W(l,d)$ for a received packet P.

{

$d \leftarrow$ destination of P

($W(l,d)$ is not updated under the following two conditions.)

if ($primary(d) = -1$) **do return**

if (P is a control packet and *indirect_routing* $= 1$) **do return**

(Select link l^* to which P is "assigned.")

if ($g(l,d) = 0$ for all l) **do** {

$l^* \leftarrow primary(d)$

}

else do {

$$l^* \leftarrow \arg \min_{l:g(l,d)>0} [\kappa W(l,d) + U(l,d) + \omega]/g(l,d)$$

}

(Add packet length to $W(l^*,d)$)

$W(l^*,d) \leftarrow W(l^*,d) + |P|$

}

Update W at Epoch()

Description: Called each epoch to recompute $W(l,d)$ based on new flows.

{

for all (d) **do** {

for all (l) **do** $W(l,d) \leftarrow 0$

$P \leftarrow$ first packet on queue d

while ($P \neq$ NULL) **do** {

Update W for Arrival(P)

$P \leftarrow$ next packet on queue d

}

}

}

Update G()
***Description:** Called each slot to increment the priority credit $G(l,k)$ for all outgoing links l and all priorities $k = 0, 1, 2$, according to the input parameters priority_k_rate and priority_k_burst.*

```
{
    for all (l and k) do {
        G(l,k) ← G(l,k) + c(l) * slot_size * priority_k_rate
        if (G(l,k) > c(l) * priority_k_burst) do {
            G(l,k) ← c(l) * priority_k_burst
        }
    }
}
```

4.3.4 Notes

- Using $W(l,d)$ instead of assigning each packet to a link preserves the advantage of using destination queues. (Assignment can be made at the last possible moment.)
- Selecting d according to arrival time (FCFS) results in fast response to changes in traffic rates, and thus improves fairness to bursty traffic.
- Since packets are enqueued in order of priority, higher-priority packets are more likely to use the primary link, if Θ's are used.
- Flow computation is independent of priorities.

4.4 Link Transmissions

This section discusses the mechanisms that cause a link to select and transmit the next packet, i.e., **Kick This Link()** and **Link Available Interrupt Handler()**.

4.4.1 Algorithms

An outgoing link l is said to be active if it is currently transmitting a packet, and inactive otherwise. **Kick This Link(l)** activates link l, if it is inactive, by scheduling an immediate Link Available Interrupt, and is called in two cases:

1. Following the epoch processing, **Kick This Link(l)** is called for all outgoing links l, since the variables that determine what packets are eligible for each link may have changed.

2. Whenever any packet is enqueued for forwarding, **Kick This Link(l)** is called for all outgoing links l, to attempt to activate a link that may be used to transmit the packet.

The **Link Available Interrupt Handler(l)** is an interrupt routine that is run when a corresponding timer expires (or, in some systems, when a device driver causes the interrupt), indicating that link l is ready to transmit a packet. The routine will attempt to obtain an eligible packet for transmission from one of the destination queues, by calling **Find Eligible Packet(l)**. If an eligible packet P is found, the packet is sent to the link interface, and **Register Transmission(l, P)** is called. If there are no eligible packets, the link becomes inactive.

Local Variables

- Active_Links is a set containing all links l that are currently transmitting packets.

4.4.2 Pseudocode

Kick This Link(l)

***Description:** Activates link l if it is inactive.*

```
{
    if (l is not in Active_Links) do {
        Add l to Active_Links.
        Schedule an immediate Link Available Interrupt for l.
    }
}
```

Link Available Interrupt Handler(l)

***Description:** An interrupt routine that is run when a corresponding timer expires (or, in some systems, when a device driver causes the interrupt), indicating that the link passed is ready to transmit a packet.*

```
{
    P ← Find Eligible Packet(l)
    if (P = NULL) do {
        Remove l from Active_Links          Link becomes inactive.
        return
    }
    Register Transmission(l, P)
    Send P to network interface
    Schedule Link Available Interrupt for l to arrive when transmission of P completes
}
```

5 Packet Processing

Separate queues are maintained for each destination. When a link becomes available, it searches the destination queues, according to its link schedule, and attempts to fetch a packet that is eligible for it (see Section 4.3). No queues are used on the links. Thus, the decision on which link a particular packet will be transmitted over is made at the last possible moment. This section describes various aspects of the STIP3 packet processing, including queueing, forwarding, and retransmission strategies.

5.1 Incoming Packets

Packet Arrival($P, j, mode$) processes every packet that is generated, timed out, or received over a link from neighbor j. The variable *mode* can be ORIGINATING if P originated at the processing node, NETWORK if P was received over a link, or TIMEOUT if the packet is a copy of a packet that has timed out. **Packet Arrival**() is called by the following procedures: **Receive Packet Interrupt Handler()**, **Process Originating Traffic()**, **Generate Update Packets()**, and **Timeout Interrupt Handler()**.

If any packet is received over a link, **Packet Arrival()** sets $jammed = 0$ to indicate that the node is not jammed.

5.1.1 Pseudocode

Packet Arrival(P, j, *mode*)
Description: Processes any packet that has been generated, timed out, or received over a link from j.
{

if (*mode* = NETWORK) **do** $jammed \leftarrow 0$
(Generate ack if appropriate.)
if (*mode* = NETWORK and $P_t \neq$ ACK) **do** **Generate Ack(P, j)**
$d \leftarrow$ destination of P
if (d = this node) **do** {
if (P_t = PROBE) **do** Discard P
if (P_t = ACK) **do** **Process Ack Final Destination(P)**
if (P_t = UPDATE) **do** **Process Update Final Destination(P)**
if (P_t = TRAFFIC) **do** **Process Traffic Final Destination(P)**
return
}
if (*mode* $\neq$ TIMEOUT) **do** $P_a \leftarrow t$ *Set time of arrival.*
(Update arrival_count(d) for computing measured incoming traffic $x(d)$.)
if ((*mode* = NETWORK or ORIGINATING) and
(P_t = TRAFFIC or *indirect_routing* = 1)) **do** $arrival\_count(d) \leftarrow arrival\_count(d) + |P|$
Update W for Arrival(P)
(Process and possibly enqueue packet for forwarding.)
if (P_t = ACK) **do** **Process Ack Packet for Forwarding(P)**
if (P_t = UPDATE) **do** **Process Update Packet for Forwarding(P, *mode*)**
if (P_t = TRAFFIC) **do** **Process Traffic Packet for Forwarding(P, *mode*)**

}

Receive Packet Interrupt Handler(l, P)
Description: Called when the network interface indicates that a packet has been received and is ready to be processed by the protocol.
{

$j \leftarrow$ originating node of link l
Packet Arrival(P, j, NETWORK)

}

Process Originating Traffic(P)

Description: Processes and enqueues a traffic packet that is originating at this node.

{

Add STIP header to P — *Gives this packet a unique sequence number for this source. See note below.*

$P_a \leftarrow t_0$ — *Set time of arrival.*

Packet Arrival(P, NULL, ORIGINATING)

}

Process Ack Final Destination(P)

Description: Processes an ack packet that has reached its final destination.

{

$l \leftarrow$ link in Xmt_List(P_x) — *Look up link used for transmission number P_x.*

Process Ack Probability(l, P_x) — *Sets ACK_RECEIVED flag in Xmt_List*

Process Ack Window(l, P_x)

Get type, source, and seq. no. of acked packet from Xmt_List.

if (acked packet is of type traffic) **do**
  Delete any queued traffic packet with same source and seq. no.

Discard P

}

Process Traffic Final Destination(P)

Description: Processes a traffic packet that has reached its final destination.

{

Strip STIP header from P and pass to user's receive handle

}

Process Update Final Destination(P)

Description: Processes an update packet that has reached its final destination.

{

Parse P

for all (j and d) **do** {

if (P contains entry for $e\_nbr(j,d)$ and $P_{\text{gen}} > e\_nbr\_time(j,d)$) **do** {

Update $e\_nbr(j,d)$

$e\_nbr\_time(j,d) \leftarrow P_{\text{gen}}$

}

if (P contains entry for $DAG_{\text{nbr}}(j,d)$ and $P_{\text{gen}} > DAG\_nbr\_time(j,d)$) **do** {

```
            Update DAG_nbr(j, d)
            DAG_nbr_time(j, d) ← P_gen
        }
    }
    Discard P
}
```

Generate Ack(P, j)

Description: Generates an ack packet in response to a traffic or update packet received over a link from j.

```
{
    Generate ack packet R with destination j
    R_a ← t                                          Set time of arrival.
    R_x ← P_x                                        Reuses transmission number.
    Enqueue(R)
    if (use_forced_acks = 1 and indirect_routing = 1) do {
        for all (l to xmitting neighbor j) do {
            if (l is idle) do                        There may be more than one if
                                                     the network has parallel links.
              Duplicate R and transmit over l        See notes below.
        }
    }
}
```

Process Ack Packet for Forwarding(P)

Description: Enqueues an ack packet for forwarding, unless it exceeds the age limit for acks.

```
{
    if (t − P_gen < ack_life) do  Enqueue(P)
    else do Discard P
}
```

Process Update Packet for Forwarding($P, mode$)

Description: Processes and possibly enqueues an update packet for forwarding.

```
{
    (Discard P if it was generated at another node and it exceeds the age limit for updates.)
    if (t − P_gen ≥ update_life and P_s ≠ this node) do {
        Discard P
        return
```

```
    }
    (Remove outdated fields from an old update generated at this node.)
    if (P_s = this node and mode ≠ ORIGINATING) do {
        Remove fields from P that disagree with local variables e(d) and SDAG(j, d)
        P_gen ← t
        if (P has no remaining fields) do Discard P and return
    }
    (Combine P with any queued update having same source and destination.)
    if (there exists a queued update R with R_s = P_s and R_d = P_d) do {
        P ← P ∪ R                                   Union is defined in notes below.
        P_gen ← max{P_gen, R_gen}
        P_a ← min{P_a, R_a}
        Discard R
    }
    Enqueue(P)
}
```

Process Traffic Packet for Forwarding($P, mode$)
Description: Processes and possibly enqueues a traffic packet for forwarding.

```
{
    d ← destination of P
    (Exercise flow control if option is selected.)
    if (flow_control = 1 and mode = ORIGINATING and
      ((use_old_flows = 1 and e(d) ≥ e_max) or
      (use_old_flows = 0 and e(d) ≥ 0.9/φ))) do {
        Discard P
        return
    }
    if (t − P_gen < traffic_life) do Enqueue(P)
    else do Discard P
}
```

5.1.2 Notes

- An "idle" link is one that is not currently being used for transmitting any packet. A possible enhancement would be to queue forced acks onto links when they are currently transmitting a probe.

- This enhancement may cause duplicate acks to be transmitted over the same link (if the normally enqueued ack is eventually transmitted over that link). Depending on the implementation, it may be desirable to avoid this duplication. For instance, if, before returning, the enqueueing process does not wake up idle links to give them a chance to grab packets, then it is possible that duplicate acks will be transmitted back to back on the same link.

- Each packet generated by a node (be it originating application traffic, or a generated update or ack packet) is given a unique sequence number for that node. Therefore, the (source, sequence number) pair within the STIP packet header uniquely identifies that packet and permits duplicate filtering to be performed by the **Enqueue()** function. We do not address wrap-around implementation issues for the sequence numbers and transmission numbers in this document.

- In **Process Update Packet for Forwarding()**, it is possible that the union $P \cup R$ of two update packets is formed. The union contains every field that occurs in either packet. For fields contained in both packets, the value from the more recently generated packet is used.

- If the flow-control option is selected, traffic packets originating at the processing node are dropped if $e(d)$ exceeds a certain value, as indicated by the pseudocode for **Process Traffic Packet for Forwarding()**. This method is experimental and has not yet been evaluated.

5.2 Enqueueing

The queues are FCFS within a set of packets of the same queuing priority. However, packets of types with higher priorities are placed ahead of all packets of lower priority. The queuing priorities are as follows, with 1 being the highest:

1. Acks in response to traffic packets
2. Acks in response to update packets
3. Acks in response to probe packets
4. Update packets
5. High-priority traffic packets
6. Low-priority traffic packets.

Queuing priorities should not be confused with scheduling priorities, which are presented in Section 4.3.2. All acks have been given higher queuing priority for two reasons: First, in order to be responsive to dynamic links, a small ack timeout window

is required. To maintain this small window, acks must be received quickly and with minimal delay variance. Second, acks are very short packets and do not add much delay to update and traffic packets. In addition, the various types of acks are ordered such that the acks that respond to the larger packets have higher priority, to avoid unnecessary retransmissions of large packets.

5.2.1 Pseudocode

Enqueue(P)
***Description:** Enqueues packets on destination-oriented queues according to their priority and time of arrival.*
{

(Filter duplicate traffic packets.)
if (packet R on queue P_d exists with $R_q = P_q$ and $R_s = P_s$) **do**
Discard P — *Packet with same sequence and source already present.*

Insert P into queue according to
queuing priority and time of arrival — *Discussed above.*

(Kick all outgoing links.)
for all (l) **do Kick This Link**(l)

}

5.2.2 Notes

- Priorities do not apply to probe and forced ack packets, since these are transmitted only over links that are inactive, and furthermore are directed over a particular link and thus would not belong in a destination queue.
- When any type of packet is enqueued, all links are kicked to attempt to start up an inactive link that may be used to transmit the packet.

5.3 Retransmissions

If no ack has been received for a traffic or update packet within the timeout window (see Section 4.2) of the link last used for transmitting the packet, it is reinserted into the destination queue, just ahead of all packets of the same type with arrival times that are later than the arrival time of the packet being retransmitted. This method preserves the FCFS priorities of all packets of the same type, and requires that arrival timestamps (P_a) be tagged to traffic and update packets as they arrive or are generated at this node.

5.3.1 Global Variables

Export variables

Xmt_List

5.3.2 Pseudocode

Register Transmission(l, P)

Description: Performs the bookkeeping to permit this transmission to be matched with a future ack, and to cause the generation of a duplicate transmission, in case no ack is received.

```
{
    if (P_t = ack) do return                    P_x is already set to Xmt_Number
                                                of packet being acked.
    P_x ← Xmt_Cnt                               Tags packet with a transmit
                                                count, so that a future ack can be
                                                matched to this transmission.
    Xmt_List(Xmt_Cnt) ← (t_0 + |P|/c(l), l, ACK_NOT_RECEIVED)
                                                Each Xmt_List() entry contains a
                                                (time, link, ack flag) triple. t_0 is
                                                the current time.
    Xmt_Cnt ← Xmt_Cnt +1
    Set timeout interrupt for P to w(l)
}
```

Timeout Interrupt Handler(P)

Description: Called when the timer set by Register Transmission() expires.

```
{
    if (Ack flag in Xmt_List(P_x) = ACK_RECEIVED) do {
        Discard P
        return
    }
    Packet Arrival(P, NULL, TIMEOUT)
}
```

5.3.3 Notes

- Note that a packet may be retransmitted on a different link from the previous transmission.
- In later implementations, it may be useful to maintain a retransmit counter to limit the number of retransmissions. However, the input parameters $traffic\_life$

and *update_life* can be used to limit the packet life and thus the number of retransmissions.

5.4 Traffic Measurements

5.4.1 Global Variables

Export variables

$q(d)$ $\bar{q}(d)$
$x(d)$ $cntrl\_flow(l)$

Import variables

Δt $arrival\_count(d)$ $cntrl\_count(l)$
β β'

5.4.2 Algorithm

This section discusses the updating of traffic measurements used for epoch processing, including the queue sizes, the arrival traffic rate for each destination, and the control traffic rate for each outgoing link.

The instantaneous queue size $q(d)$ is the sum in bits of all packets (ack, traffic, and update) in the queue for destination d awaiting transmission or retransmission. Packets awaiting acks are not counted in the computation of $q(d)$ before their retransmit timeout expires. In addition, if $indirect\_routing = 0$, so that updates and acks are sent only on the direct link to their destination, then updates and acks are not counted in $q(d)$. The average queue size $\bar{q}(d)$ is the average of $q(d)$ over all time slots since the last epoch.

The variable $arrival\_count(d)$ counts the amount (in bits) of d-traffic either received over a link or generated since the last epoch, excluding control traffic if $indirect\_routing = 0$. This variable is updated in **Packet Arrival()**. The variable $x(d)$ is the smoothed arrival rate, and is updated each epoch using $arrival\_count(d)$. The smoothing constant for $x(d)$ is β. (The update equations are given in the pseudocode.)

In addition, $x\_count(d)$ counts the number of epochs that have passed sinced the last time $arrival\_count(d)$ was nonzero. If $x\_count(d)$ exceeds some constant (we used the number of epochs per second), then $x(d)$ is set to zero, to indicate that no traffic has arrived recently.

The variable $cntrl\_count(l)$ counts the amount (in bits) of control traffic that has been sent over link l since the last epoch. This variable is updated in **Find Eligible Packet()**. The variable $cntrl\_flow(l)$ is the smoothed rate of control traffic over link l, and is updated each epoch using $cntrl\_count(l)$. The smoothing constant for $cntrl\_flow(l)$ is β'.

5.4.3 Pseudocode

Update Measurements()

Description: Called by **Epoch Processing()** *once per epoch to update* $\bar{q}(d)$, $x(d)$, *and* $cntrl\_flow(l)$.

{

for all (d) **do** {

(Update $x(d)$*.)*

$temp \leftarrow arrival\_count(d)/\Delta t$

if ($temp = 0$) **do** $x\_count(d) \leftarrow x\_count(d) + 1$

else do $x\_count(d) \leftarrow 0$

$x(d) \leftarrow (1 - \beta)x(d) + \beta\ temp$

if $(x\_count(d) > 1/\Delta t)$ **do** $x(d) \leftarrow 0$

$arrival\_count(d) \leftarrow 0$ — *Reset arrival count for next epoch.*

(Update $\bar{q}(d)$*.)*

$slots\_per\_epoch \leftarrow \Delta t/slot\_size$

$qbar(d) \leftarrow q\_sum(d)/slots\_per\_epoch$

$q\_sum(d) \leftarrow 0$ — *Reset queue sum for next epoch.*

}

for all (l) **do** {

(Update $cntrl\_flow(l)$*.)*

$temp \leftarrow cntrl\_count(l)/\Delta t$

$cntrl\_flow(l) \leftarrow (1 - \beta')cntrl\_flow(l) + \beta'\ temp$

$cntrl\_count(l) \leftarrow 0$ — *Reset control count for next epoch.*

}

}

Upd..e Queue Sum()

Description: Called each slot by **Epoch Processing()** *to compute* $q\_sum(d)$, *which is used for computing* $q(d)$.

{

for all (d) **do** $q\_sum(d) = q\_sum(d) + q(d)$

}

5.4.4 Note

- An alternative implementation for updating $\bar{q}$ would be to update it whenever an event changes the current queue size.
- In general, whenever $indirect\_routing = 1$, update and ack packets are treated like traffic packets for routing and flow computation [and are therefore counted in $q(d)$ and $x(d)$].

6 Network Algorithms

6.1 Link Delay

This section presents the computation of the raw link delay $v(l)$, the expected delay $D(l,d)$ for a packet to reach destination d if it uses link l, the primary link $primary(d)$, and the effective link capacity $c'(l)$.

6.1.1 Global Variables

Export variables

$v(l)$	$c'(l)$
$D(l,d)$	$primary(l)$

Import variables

$p(l)$	$w(l)$	$c(l)$
$u(l)$	$\bar{P}$	

6.1.2 Algorithm

The raw delay of a link $v(l)$ is its delay due to potential retransmissions, propagation, and the time required to transmit a packet; and is computed as follows:

$$v(l) \leftarrow w(l)(1/p(l) - 1) + u(l) + \bar{P}/c(l) + link\_bias \ .$$

A large value for the link bias encourages routing on minimum-hop paths, which may improve stability in some situations. The expected delay $D(l,d)$ is then computed as $v(l) + e\_nbr(j,d)$, where j is the tail of l.

The primary link $primary(d)$ for destination d is defined to be the outgoing link l in $DAG(d)$ with the minimum value of $D(l,d)$. If there does not exist such a link with finite delay, we set $primary(d) = -1$ to indicate that no primary link exists for d.

We define the effective capacity by $c'(l) = p(l)c(l)$ if $indirect\_routing = 1$, and by $c'(l) = p(l)[c(l) - cntrl\_flow(l)]$ if $indirect\_routing = 0$. In the latter case, control packets are always sent on the direct link to their destination and thus do not require routing. We therefore choose, in that case, to subtract the measured control traffic

from the link capacity, while ignoring control traffic when computing $x(d)$, $\bar{q}(d)$, and the flows $f(l,d)$.

6.1.3 Pseudocode

Update Link Delays()
Description: Updates $v(l)$, $D(l,d)$, primary(d), and $c'(l)$.

```
{
    for all (l) do {
        j ← tail node of link l
        v(l) ← w(l)(1/p(l) − 1) + u(l) + P̃/c(l) + link_bias
        D(l,d) ← v(l) + e_nbr(j,d)
        (Compute effective link capacity.)
        if (indirect_routing = 0) do
          c'(l) ← p(l)[c(l) − cntrl_flow(l)]
        else do c'(l) ← p(l)c(l)
    }
    for all (d) do {
        (Compute primary link for d.)
        min_delay ← ∞
        primary(d) ← −1
        for all (l) do {
            if (l ∈ DAG(d) and D(l,d) < min_delay) do {
                min_delay ← D(l,d)
                primary(d) ← l
            }
        }
    }
}
```

6.2 Computing the Link Flows

From $\bar{q}(d)$, $x(d)$, $p(l)$, $c'(l)$, and $D(l,d)$ for all d and l, each node updates the flows $f(l,d)$, and the offered flows $g(l,d)$ for all destinations d and links l.

6.2.1 Global Variables

Export variables

$f(l,d)$ $g(l,d)$

Import variables

$\bar{q}(d)$	$x(d)$	$p(l)$
$c'(l)$	$D(l,d)$	ϕ

6.2.2 Algorithm

STIP3 allows the option of using either the algorithm of STIP2 or its own new algorithm for computing flows. Both methods try to minimize predicted near-future queuing delay, but the STIP2 method minimizes a local objective, while the STIP3 method uses marginal cost to minimize a global objective. In the STIP2 method, $e(d)$ represents expected delay, while in the STIP3 method, $e(d)$ r[illegible]rginal cost.

STIP2 Flow Computation

The flow variables $f(l,d)$ are computed, using gradient descent, so as to minimize the local objective function

$$J = \sum_d [e'(d)]^M + \sigma \frac{\sum_l f(l,d)^2}{[\sum_l f(l,d)]^2}.$$

subject to

$$\sum_d f(l,d) \leq c'(l) ,$$

where σ is the flow spreading parameter and $M \geq 1$. A large value of σ encourages the spreading of traffic over multiple paths. Varying M achieves a tradeoff between minimizing the sum of the delays and minimizing the maximum delay.

The expected delay $e'(d)$ is the sum of local queuing delay and path delay:

$$e'(d) = Q_d + L_d ,$$

$$Q_d = \frac{1}{y(d)} \int_0^\infty \phi e^{-\phi t} [\bar{q}(d) + t\{x(d) - y(d)\}]_+ dt ,$$

$$L_d = \frac{1}{y(d)} \sum_l f(l,d) D(l,d) ,$$

where $1/\phi$ represents the average time into the future over which we wish to minimize delay; the notation $[x]_+$ denotes the positive part of x; and $y(d)$ is the sum of the outgoing flows for destination d, i.e.,

$$y(d) = \sum_l f(l,d) .$$

STIP3 Flow Computation

We define $x(i,d)$ to be the variable $x(d)$ maintained by node i, and we define $y(i,d)$, $z(i,d)$, $\bar{q}(i,d)$, and $f(i,l,d)$ similarly. For the STIP3 algorithm, the goal is to minimize the global objective

$$J = \sum_{i \in V} \sum_{d \neq i} J_{i,d}$$

subject to the link capacity constraints, where

$$J_{i,d} = Q_{i,d} + L_{i,d} ,$$

$$Q_{i,d} = \int_0^\infty \phi e^{-\phi t}[\bar{q}(i,d) + \{x(i,d) - y(i,d)\}t]_+ dt ,$$

$$L_{i,d} = \sum_l \{f(i,l,d)v(l) + \sigma[f(i,l,d)]^2\} .$$

As with STIP2, a large value of σ encourages spreading over multiple paths.

The queuing delay $Q_{i,d}$ is a function of the flow surplus $z(i,d) = x(i,d) - y(i,d)$ and is given by

$$Q_{i,d}(z(i,d)) = \bar{q}(i,d) + \frac{z(i,d)}{\phi}[1 - I\{z(i,d) < 0\}e^{\phi\bar{q}(i,d)/z(i,d)}] ,$$

where $I\{z < 0\}$ is 1 if $z < 0$ and 0 otherwise.

It is easy to show that the above problem reduces to the classical convex flow problem defined on a modified network in which an imaginary surplus link is added from i to d. The surplus $z(i,d)$ then becomes a link flow on the surplus link from i to d, with cost $Q_{i,d}(z(i,d))$. Applying optimization theory (e.g., [1, 2]) to this reduced problem, we obtain the following optimality conditions.

Lemma 1 The flows $f(i,l,d)$ are optimal if and only if there exist numbers $e(i,d)$ for each (i,d) pair, and numbers $R(l) \leq 0$ for each link l from i to j, such that the following equations are satisfied:

$$\begin{aligned} e(i,d) &= e(j,d) + v(l) + 2\sigma f(i,l,d) - R(l) , \quad f(i,l,d) > 0 , \\ e(i,d) &\leq e(j,d) + v(l) + 2\sigma f(i,l,d) - R(l) , \quad f(i,l,d) = 0 , \\ e(i,d) &= Q'_{i,d}(z(i,d)) , \end{aligned}$$

where $R(l) = 0$ if $\sum_d f(i,l,d) < c'(l)$. Note that the third equation, which corresponds to the surplus links, can be substituted into the first two equations to eliminate $e(i,d)$.

The above convex flow problem can be solved using the distributed relaxation method [1]. In one iteration of the relaxation method (i.e., one epoch), each node i recomputes $e(i,d)$, $R(l)$, and $f(i,l,d)$ to satisfy the above optimality conditions at node i and the flow conservation equation $z(i,d) = x(i,d) - y(i,d)$. The question now is how to solve these equations. The STIP3 method is based on the following result.

Theorem 1 The flows satisfy the optimality conditions of Lemma 1 if and only if they solve the following local problem for each node i:

$$\text{minimize} \quad \sum_d [Q_{i,d} + L^+_{i,d}] \tag{6.1}$$

where

$$L^+_{i,d} = \sum_l \{f(i,l,d)D(l,d) + \sigma[f(i,l,d)]^2\} . \tag{6.2}$$

Note that the only difference between $L_{i,d}$ and $L^+_{i,d}$ is that the link delay $v(l)$ has been replaced with $D(l,d)$, which is the expected delay to d using link l.

In light of the above theorem, we can perform the relaxation method to solve the global problem, by repeatedly solving the local problem of Theorem 1. This can be done using gradient descent, as in STIP2. The details of the algorithm are given in the pseudocode.

6.2.3 Pseudocode

Update Flows()

Description: *Updates the flows $f(l,d)$ and $g(l,d)$ based on the link delays $D(l,d)$, neighbor delays $e\_nbr(l,d)$, queue sizes $\bar{q}(d)$, and measured arrival rates $x(d)$.*

```
{
    (Set appropriate flows to zero.)
    for all (l and d) do {
        if (x(d) + q̄(d) = 0 or l ∉ DAG(d)) do  f(l,d) ← 0
    }
    for all (l) do {
        (Scale flows for l if necessary so as not to exceed new c'(l).)
        flow_sum(l) ← Σ_d f(l,d)
        if (flow_sum(l) > c'(l)) do

        for all (d) do f(l,d) ← f(l,d)c'(l)/flow_sum(l)
    }
    while (iteration < num_steps and flag = 0) do {
                                          flag = 1 implies flows converged.
        Iterate Flows()                   Does one iteration of gradient de-
                                          scent.
    }
    for all ((l,d) : l ∈ DAG(d)) do {
        g(l,d) ← f(l,d)/p(l)
    }
}
```

Iterate Flows()

Description: Does one iteration of gradient descent on flows.

{

for all (d) **do** {

(Compute the gradient $J'(l,d)$ for STIP2 or STIP3.)

if $(use\_old\_flows = 1)$ **do** $J'(l,d) \leftarrow$ **Compute Gradient2**(l,d)

if $(use\_old\_flows = 0)$ **do** $J'(l,d) \leftarrow$ **Compute Gradient3**(l,d)

}

for all (l) **do** {

if $(f(l,d') = 0$ for all $d')$ **do** $R(l) \leftarrow 0$

else do $R(l) \leftarrow$ average of $J'(l,d')$ over d' s.t. $f(l,d') > 0$

}

(member(l) is the set of d such that $f(l,d)$ can change.)

for all (l) **do** $member(l) \leftarrow \{d : f(l,d) > 0\}$

(Compute most desired link for each d.)

for all (l) **do** $flow\_sum(l) \leftarrow \sum_d f(l,d)$

for all (d) **do** {

for all (l) **do** {

if $(flow\_sum(l) < c'(l))$ **do** $desirability(l,d) \leftarrow -J'(l,d)$

if $(flow\_sum(l) = c'(l))$ **do** $desirability(l,d) \leftarrow R(l) - J'(l,d)$

}

$most\_desired(d) \leftarrow l \in DAG(d)$ with max $desirability(l,d)$

Add d to $member(l)$, where $l = most\_desired(d)$

}

(Compute flow changes.)

$flag \leftarrow 1$ *Will remain 1 if flows have converged.*

$multiplier \leftarrow 1$

for all (l) **do** {

if $(use\_old\_flows = 1)$ **do** $multiplier \leftarrow c'(l)$

for all $(d \in member(l))$ **do** {

if $(flow\_sum(l) < c'(l)$ or $J'(l,d) > 0)$ **do**
$change = -step * multipler * J'(l,d)$

else if $(flow\_sum(l) = c'(l)$ and $J'(l,d) < R(l))$ **do**
$change = -step * multipler * [J'(l,d) - R(l)]$

(Limit magnitude of change to less than $max\_step * c'(l)$)

if $(|change| > max\_step * c'(l))$ **do**
$change \leftarrow max\_step * c'(l) * change/|change|$

if $(|change| > \rho)$ **do** $flag \leftarrow 0$ *Not yet converged.*

$f(l,d) \leftarrow f(l,d) + change$ *Change flows.*

if $(f(l,d) < 0)$ **do** $f(l,d) \leftarrow 0$ *Set neg. flows to zero.*

```
        }
    }
    for all (l) do {
        (Scale new flows for l if necessary so as not to exceed c'(l).)
        flow_sum(l) ← Σ_d f(l,d)
        if (flow_sum(l) > c'(l)) do
        for all (d) do f(l,d) ← f(l,d)c'(l)/flow_sum(l)
    }
}
```

Compute Gradient2(l,d)

Description: Computes the gradient $J'(l,d)$ of the STIP2 objective, assuming $M = 1$.

{

$$grad \leftarrow -\frac{x^2(d)}{\phi^2} - \frac{x(d)\bar{q}(d)}{\phi} - \bar{q}^2(d)$$
$$+I\{x(d) < y(d)\}\frac{x(d)}{\phi^2}\left[\frac{\phi y(d)\bar{q}(d)}{y(d)-x(d)} + x(d)\right]e^{-\phi\bar{q}(d)/[y(d)-x(d)]}$$
$$+\frac{y(d)D(l,d) - \sum_m f(m,d)D(m,d)}{y^2(d)}$$
$$+2\sigma\frac{y(d)f(l,d) - \sum_m [f(m,d)]^2}{y^3(d)}$$

return *grad*

}

Compute Gradient3(l,d)

Description: Computes the gradient $J'(l,d)$ of the STIP3 objec

{

(Assume a minimum queue size for d with traffic.)

if $(x(d) > 0)$ **do** $Q \leftarrow \max\{\bar{q}(d), qmin\}$

$$grad \leftarrow D(l,d) - \frac{1}{\phi}\left[1 - I\{x(d) < y(d)\}e^{-\phi Q/[y(d)-x(d)]}(1 - \frac{\phi Q}{y(d)-x(d)})\right]$$
$$+\sigma\frac{f(l,d)}{c'(l)}$$

return *grad*

}

6.2.4 Notes

- If direct routing of acks and updates is used, the measured control traffic is subtracted from the capacity of each link.
- To allow a larger step size for gradient descent, we need to bound the second derivative of the local objective. This can be achieved by assuming $\bar{q}(d) \geq qmin$ for some $qmin > 0$. (However, making $qmin$ too large will create the illusion of congestion.) STIP3 uses $qmin = 1000$ bits, the size of a typical traffic packet.
- In **Iterate Flows()**, when the flow changes are computed, the conditions $J'(l,d) > 0$ and $J'(l,d) < R(l)$ may not be needed, but were observed to help in STIP2.

6.3 Distance Variables

This section describes how the distance variables $e\_priv(d)$ and $e(d)$ are updated.

6.3.1 Global Variables

Export variables

$e(d)$

Import variables

$f(l,d)$	α_{inc}	α_{dec}
$D(l,d)$	$x(d)$	$\bar{q}(d)$

6.3.2 Algorithm

After updating its flows $f(l,d)$, each node recomputes its instantaneous distance variables $e'(d)$ based on the new flows, $\bar{q}(d)$, $x(d)$, and the link delays $D(l,d)$. If the input parameter $use\_old\_flows = 1$, indicating that the STIP2 method for computing flows is used, then $e'(d)$ represents the expected delay (including local queuing delay) to d. If the STIP3 method is used, then $e'(d)$ represents the marginal cost for d, that is, the derivative of the global cost with respect to an increase in d-traffic. The equations for $e'(d)$ are given in the pseudocode functions **Expected Delay()** and **Marginal Cost()**. The maximum allowed value for $e'(d)$ is $emax$.

The variables $e\_priv(d)$ are time-smoothed versions of $e'(d)$, using the smoothing parameters α_{inc} and α_{dec} (depending on whether $e\_priv(d)$ increases or decreases). In addition, to ensure that routing loops are avoided after the flows converge, $e\_priv(d)$ is increased if necessary, to ensure that $e\_priv(d) > e\_nbr(j,d)$ if $f(l,d) > 0$ for some link l to j. This is done in **Update Private Distance Variables()**.

If $e\_priv(d)$ has changed by a significant amount (determined by the input parameters δ and ϵ), and/or if other conditions are satisfied (see the pseudocode), then

$e(d)$ is updated to $e\_priv(d)$ and is reported to neighbors in update messages. The variables $e(d)$ are updated in **Update Reported Distance Variables()**), and update packets are created in **Generate Update Packets()**.

The variable $epochs\_since\_last\_update(d)$ keeps track of the number of epochs that have passed since the last time $e(d)$ was updated, and $epochs\_since\_last\_update$ keeps track of the number of epochs since $e(d)$ was updated for *any* d. The variables $e(d)$ are updated only if $short\_update\_period$ epochs have passed since the last update. This enforces a lower bound on the time between updates, in order to reduce update traffic. In addition, certain low-priority updates (e.g., updates that involve destinations d with no traffic and that are not required to maintain a loop-free path to d) are done only if at least $long\_update\_period$ epochs have passed since $e(d)$ was last updated.

At node i, $e(i)$ represents the delay from i to itself and so is normally zero. If $jammed = 1$, indicating that node i is jammed, then $e(i)$ is set to $emax$, to inform neighbors when i is jammed.

6.3.3 Pseudocode

Update Private Distance Variables()

Description: Called each epoch to update the smoothed local distance variables e_priv(d).

{

for all (d) **do** {

if $(use\_old\_flows = 1)$ **do** $e'(d) \leftarrow$ **Expected Delay**(d)

if $(use\_old\_flows = 0)$ **do** $e'(d) \leftarrow$ **Marginal Cost**(d)

(emax is the max value of any distance variable.)

if $(e'(d) > emax)$ **do** $e'(d) \leftarrow emax$

if ($jammed = 1$ and $d =$ this node) **do** $e'(d) \leftarrow emax$

if ($jammed = 0$ and $d =$ this node) **do** $e'(d) \leftarrow 0$

(Smooth e_priv(d) using α_{inc} if it increased)

if $(e\_priv(d) < e'(d) < emax)$ **do**
$e\_priv(d) \leftarrow (1 - \alpha_{\text{inc}})e\_priv(d) + \alpha_{\text{inc}}e'(d)$

(Smooth e_priv(d) using α_{dec} if it decreased)

else if $(e'(d) < e\_priv(d) < emax)$ **do**
$e\_priv(d) \leftarrow (1 - \alpha_{\text{dec}})e\_priv(d) + \alpha_{\text{dec}}e'(d)$

for all (nbrs j such that $f(l,d) > 0$ or $l = primary(d)$ for some l to j) **do** {

(Make sure $e\_priv(d) > e\_nbr(j,d)$.)

if $(e\_priv(d) \leq e\_nbr(j,d))$ **do**
$e\_priv(d) \leftarrow e\_nbr(j,d) + u(l) + \bar{P}/c(l) + link\_bias$

}

(e_priv(d) should not be greater than emax.)

if $(e\_priv(d) > emax)$ **do** $e\_priv(d) \leftarrow emax$

}

}

Update Reported Distance Variables()

Description: Updates the official (reported) distance variables e(d) by setting them equal to e_priv(d) under certain conditions.

{

increment *epochs_since_last_update*

for all (d) **do** increment *epochs_since_last_update*(d)

(Update only if short_update_period epochs have passed since last update.)

if $(epochs\_since\_last\_update < short\_update\_period)$ **do return**

for all (d) **do** {

(Compute the average one-hop delay Δe.)

$y(d) \leftarrow \sum_l f(l,d)$

if $(y(d) > 0)$ **do** $\Delta e \leftarrow [1/y(d)] \sum_l f(l,d)[e\_priv(d) - e\_nbr(j,d)]$

else do $\Delta e \leftarrow 0$

$tolerance \leftarrow \max\{\delta\Delta e, \epsilon\}$

(Compute time since last update for e(d).)

$\tau \leftarrow epochs\_since\_last\_update(d) * \Delta t$

$tolerance \leftarrow e^{-\mu\tau} tolerance$

if $(|e\_priv(d) - e(d)| > tolerance)$ **do** $tolerance\_exceeded \leftarrow 1$

else do $tolerance\_exceeded \leftarrow 0$

$e\_change(d) \leftarrow 0$ *Will be set to 1 if e(d) is changed.*

(Update e(d) under the following conditions.)

if ($e\_priv(d) = emax$ or $e(d) = emax$ or ($tolerance\_exceeded = 1$ and
$(x(d) + \bar{q}(d) > 0$ or *Implies d has traffic.*
$epochs\_since\_last\_update(d) \geq long\_update\_period$ or
$(e(d) \geq 1/\phi$ and $use\_old\_flows = 0)))$) **do** {

if ($e\_priv(d) \neq e(d)$) **do** {

$e(d) \leftarrow e\_priv(d)$

$e\_change(d) \leftarrow 1$

$epochs\_since\_last\_update(d) \leftarrow 0$

$epochs\_since\_last\_update \leftarrow 0$

}

}

}

}

Expected Delay(d)

Description: Computes the expected delay to destination d for STIP2.

{

if $(|DAG(d)| = 0)$ **do return** ∞

$y(d) \leftarrow \sum_l f(l,d)$

$B(d) \leftarrow \sum_l f(l,d) D(l,d)$

$$delay \leftarrow \frac{1}{y(d)}[\bar{q}(d) + B(d) - \frac{y(d) - x(d)}{\phi}][1 - I\{x(d) < y(d)\} e^{-\phi\bar{q}(d)/[y(d)-x(d)]}]$$

I_A is 1 if A is true and 0 otherwise.

return *delay*

}

Marginal Cost(d)
***Description:** Computes the marginal cost to destination d for STIP3.*
{

if ($|DAG(d)| = 0$) **do return** ∞
(Compute the gradient $J'(l,d)$ for STIP3.)
$J'(l,d) \leftarrow$ **Compute Gradient3**(l,d)
(Compute average gradient $R(l)$.)
for all (l) **do** {
if ($f(l,d') = 0$ for all d') **do** $R(l) \leftarrow 0$
else do $R(l) \leftarrow$ average of $J'(l,d')$ over d' s.t. $f(l,d') > 0$
}
(Compute minimum cost over all links in DAG.)
mincost $\leftarrow \infty$
for all ($l \in DAG(d)$) **do** {
if ($f(l,d) = c'(l)$) **do** *cost* $\leftarrow \infty$
else do *cost* $\leftarrow D(l,d) + \sigma \frac{f(l,d)}{c'(l)} - R(l)$
if (*cost* < *mincost*) **do** *mincost* $\leftarrow$ *cost*
}
return *mincost*

}

6.3.4 Notes

- By replacing $f(l,d)$ with $g(l,d)p(l)$ wherever it appears in the code, we can remove the requirement of storing an additional l by d array at the cost of additional computation.

6.4 Maintenance of the Directed Acyclic Graphs

To help avoid routing loops, all flows for each destination d are constrained to a DAG directed toward d. This section discusses the algorithm used to maintain this DAG.

6.4.1 Global Variables

Export variables
$DAG(j,d)$ $SDAG(j,d)$

Import variables

$e(d)$	$e\_nbr(j,d)$
$DAG\_nbr(j,d)$	$f(l,d)$

6.4.2 Algorithm

STIP3 maintains two DAGs, represented by the variables $DAG(j,d)$ and $SDAG(j,d)$, where j is a neighbor and d is the destination. We let $DAG(d)$ and $SDAG(d)$ denote the DAGs for destination d. If $DAG(j,d) = 1$, then all links from the processing node i to neighbor j are in $DAG(d)$; and if $DAG(j,d) = 0$, then no links from i to j are in $DAG(d)$. Therefore, even though there may be parallel links from i to j, either all of them or none of them are in the DAG, and so a DAG equivalently consists of ordered pairs (i,j).

At each epoch, $DAG(d)$ is first updated, before the flows $f(l,d)$ are updated. The flows are then allowed to be positive only on links that are in $DAG(d)$. $SDAG(j,d)$ is then set to 1 only for neighbors j such that either $l = primary(d)$ or $f(l,d) > 0$ for some link l going to j. (It follows that $SDAG(d)$ is a subgraph of $DAG(d)$.)

If $SDAG(j,d)$ changes at node i, the new value is sent in an update message to j, who then uses this information to decide whether to add (j,i) to its $DAG(d)$. The general rule is that if (i,j) is in node i's $SDAG(d)$, then node j cannot add (j,i) to its $DAG(d)$. Conflicts in which (i,j) and (j,i) are simulataneously added to the DAG are resolved according to node ID.

If a neighbor d is reported to be jammed (indicated by $e\_nbr(d,d) = emax$), then we set $DAG(d,d) = 1$ and $DAG(j,d) = 0$ for all other neighbors j. That is, we remove all links from $DAG(d)$ except direct links to d. This is done to avoid a futile attempt to send packets to d along indirect paths.

To hasten recovery from link failures, if a neighbor reports an $e\_nbr(j,d)$ of $emax$, all links to that neighbor are immediately removed from the DAG.

6.4.3 Pseudocode

Update DAG()

Description: Called by **Epoch Processing()** *to update* $DAG(j,d)$.

```
{
    for all (d) do {
        for all (neighbors j) do {
            if (DAG_nbr(j,d) = 0 and e_nbr(j,d) < emax) do
              DAG(j,d) ← 1
            else if (DAG_nbr(j,d) = 1 and SDAG(j,d) = 1
              and ID(this node) > ID(j)) do
              DAG(j,d) ← 1                          Resolve conflict using node ID.
            (If d is a jammed neighbor, then include only d in DAG.)
            if (d is a neighbor and e_nbr(d,d) = emax) do {
                if (j = d) do  DAG(j,d) = 1
                else do DAG(j,d) = 0
            }
        }
    }
}
```

Update SDAG()

Description: Called by **Epoch Processing()** *to update* $SDAG(j,d)$.

```
{
    (Update only if short_update_period epochs have passed since last update.)
    if (epochs_since_last_update < short_update_period) do return
    for all (d) do {
        for all (neighbors j) do {
            if (f(l,d) > 0 or l = primary(d) for some l to j) do  temp = 1
            else do temp = 0
            if (temp ≠ SDAG(j,d)) do {
                SDAG(j,d) ← temp
                SDAG_change(j,d) ← 1                Indicates variable has changed.
            }
            else do SDAG_change(j,d) ← 0
        }
    }
}
```

6.4.4 Note

- Initially, $DAG(j,d) = 1$ for $j = d$ and 0 otherwise; i.e., $DAG(d)$ includes only direct links to d.

6.5 Update Messages

When the variables $e(d)$ and $SDAG(j,d)$ are changed, as indicated by the bit variables *e_change*(d) and *SDAG_change*(j,d), they are sent to neighbors in update messages. Each variable $SDAG(j,d)$ is sent only to neighbor j.

To reduce the number of update messages, $e(d)$ is not sent from a node i to neighbors j such that $SDAG(j,d) = 1$. Such a neighbor has no need for this information, since it is not allowed to route through node i to destination d. However, as soon as $SDAG(j,d)$ changes to 0, $e(d)$ is sent to j.

6.5.1 Pseudocode

Generate Update Packets()
Description: *If there have been any changes to $e(d)$ or $SDAG(j,d)$, create and enqueue update packets.*

```
{
    for all (neighbors j) do {
        for all (d) do {
            if ([e_change(d) = 1 and SDAG(j,d) = 0] or
              [SDAG(j,d) = 0 and SDAG_change(j,d) = 1]) do {
                Add new e(d) to the Update packet U(j) for j
            }
            if (SDAG_change(j,d) = 1) do
              Add new SDAG(j,d) to the Update packet U(j) for j
        }
    }
    for all (Update packets U(j) (if any)) do
      Packet Arrival(U(j), NULL, ORIGINATING)          Section 5.1
}
```

6.6 Queue Thresholds

This section shows how the queue thresholds $\Theta(l,d)$ are computed. These thresholds are used by the link scheduling algorithm (Section 4.3) for selecting a packet from one of the destination queues.

6.6.1 Global Variables

Export variables

$\Theta(l,d)$

Import variables

$f(l,d)$ $D(l,d)$

6.6.2 Algorithm

For each (l,d) pair, the threshold $\Theta(l,d)$ is chosen so that a bit in that position in the d-queue would have the same expected delay (to d) whether it used link l immediately or waited in the queue to use the best link. Therefore, a packet that lies entirely above position $\Theta(l,d)$ would minimize its delay by using l immediately instead of waiting for a better link.

Once the flows are computed, the thresholds are computed as follows. First, the outgoing links are ordered according to delay. For each d, we let $l_k(d)$ be the link in $DAG(d)$ with the kth smallest delay $D(l,d)$. In particular, $l_1(d) = primary(d)$.

Let $\Theta_k = \Theta(l_k(d),d)$. Then clearly $\Theta_1 = 0$, and

$$\Theta_{k+1} = \Theta_k + [D(l_{k+1}(d),d) - D(l_k(d),d)] \sum_{j=1}^{k} f(l_j(d),d) \quad .$$

6.6.3 Pseudocode

Update Queue Thresholds()

Description: Updates the $\Theta(l,d)$ queue thresholds, based on the current delays $D(l,d)$ and flows $f(l,d)$. Initially, $\Theta(l,d) = 0$.

```
{
    (If use_thetas = 0, then Θ(l,d) = 0 for all l,d.)
    if (use_thetas = 0) do return
    for all (d) do {
        if (|DAG(d)| = 0) do return                    No links in DAG(d).
        for all (k = 1,2,...,|DAG(d)|) do
          l_k(d) ← link l ∈ DAG(d) with kth smallest D(l,d)
        Θ(l_1(d),d) ← 0
        for all (k = 2,3,...,|DAG(d)|) do {
            Θ(l_k(d),d) ← Θ(l_{k-1}(d),d) +
              [D(l_k(d),d) − D(l_{k-1}(d),d)] Σ_{j=1}^{k-1} f(l_j(d),d)
        }
    }
}
```

6.6.4 Notes

- Note that $\Theta(l, d)$ may be larger than the queue size $q(d)$ for some l. In this case, the scheduling algorithm will select the last packet in the queue.
- The following is an example of the (greedy) reasoning used to decide whether a packet waiting in a queue for destination d should use link l immediately, or wait for a better link. Suppose $\Theta(l, d)$ is 25. This means that if there are 20 bits ahead of a particular packet, then it is better for the packet to wait for a better link. If there are 25 bits ahead of the packet, it is just as good to use link l as it is to wait for a better link. If there are 30 bits ahead of the packet, then the packet should use link l.
- Since $\Theta(l_1(d), d) = 0$, the first packet in the d-queue can use the primary link for d. This is because the number of bits ahead of it is at least zero. A packet can use link l if the bit sum of all the packets ahead of it is at least $\Theta(l, d)$.
- Packets that have been transmitted but not yet acked or timed out are not counted in any of the queue size variables. Thus, they also do not affect the comparisons of queue size to a threshold.

7 NAPI+ Implementation Notes

This section describes approximations and requirements, and contains other notes pertinent to the NAPI+ simulation of STIP3.

7.1 Limitations of the Simulation

The following are limitations of the NAPI+ simulation of STIP3.

- Although the protocol permits multiple parallel links between pairs of nodes, the NAPI+ simulation permits at most one link between any pair of nodes. However, extension to multiple parallel links would not be difficult.

- The NAPI+ simulation does not simulate processing time. Therefore, epoch processing and packet processing are not taken into account. This is reasonable for the model considered during the EDMUNDS project, where we target links with rates of less than 10 Mbps and we assume that 100-MIP, high-performance processing can be dedicated to the job of epoch processing. However, for accurate simulations in networks with higher-capacity links (e.g., 100 Mbps), a schedule would be required to perform epoch processing in a way that would approximate the behavior of an actual high-performance packet switch.

- Probabilistic packet loss is not currently modeled in the simulation. Thus, all packets transmitted over "down" links are lost, and all packets transmitted over "up" links are successful. Packet success or failure is determined at transmission time. However, the rate at which links can be brought up and down can be arbitrarily fast.

7.2 Mapping of Parameters Between Document and Environment File

Table 7.1 gives the names used in the simulation environment file for the input parameters listed in Section 2.2.

Table 7.1: Mapping of Variable Names Between Document and Source Code

Document	Environment File	Default Value
Δt	s3_epoch_interval	0.2
δ	s3_tolerance	0.25
ϵ	s3_epsilon	0.15
μ	s3_epsilon_decay	0.0
σ	s3_spread	0.2
spread_DAG	s3_spread_DAG	1
indirect_routing	s3_indirect_routing	0
link_bias	s3_link_bias	0.0
M	s3_PWR	1
W_B	env_t_mad_scalar	0.2
W_M	env_max_t_scalar	4.0
W_S	env_sigma	0.8
W_T	env_t_scalar	3.0
α_{inc}	s3_alpha_inc	0.5
α_{dec}	s3_alpha_dec	1.0
β	s3_beta	0.5
β'	s3_cntrl_beta	0.25
η	s3_eta	4.0
γ	env_gamma	1.0
ϕ	s3_phi	0.2
emax	s3_emax	10.0
step	s3_step	1.0
max_step	s3_max_inc	0.1
num_steps	s3_num_steps	30
ρ	s3_rho	0.01
κ	s3_W_coeff	1.0
flow_control	s3_flow_control	0
priority_k_rate	s3_priority_k_rate	1.0
priority_k_burst	s3_priority_k_burst	1.0
$\bar{P}$	s3_avg_packet_size	1000
auto_gamma	s3_auto_gamma	0
use_old_flows	s3_use_old_flows	0
use_thetas	s3_use_theta	1
ack_life	s3_ack_life	1.0
update_life	s3_update_life	1.0
traffic_life	s3_traffic_life	5.0
detect_jam	s3_detect_jam	1
short_update_period	s3_short_update_period	1
long_update_period	s3_long_update_period	10
probe_interval	s3_probe_interval	0.19
use_forced_acks	s3_use_forced_acks	0

Notes

- When *s3_old_flows* is set to 1, the default value for *s3_spread_DAG* is 0.
- It is recommended that *s3_W_coeff* be set to a large number, such as 100.
- For better convergence, it is recommended that *s3_num_steps* be set to a larger number than indicated (e.g., 300) and *s3_step* be set to a smaller number (e.g., 0.1).
- It is recommended that *s3_phi* be set to 1 when *s3_old_flows* is 1.

7.3 Recent Code Changes and Remaining Bugs

The 29 July 1993 version of the STIP3 simulation was the version used for most of the experiments presented in this report. After running the experiments presented in this report, we made some minor changes to STIP3. Therefore, the STIP3 code used for the experiments differs from the presented pseudocode in the four ways listed below.

The first two changes were incorporated into the 15 August 1993 version of the STIP3 simulation code

1. In the pseudocode procedure **Update Reported Distance Variables**, the condition ($e(d) \geq 1/\phi$ and $use\_old\_flows = 0$) was not included in the "if" statement that decides whether to update $e(d)$. This omission has been corrected beginning with the 15 August 1993 version of the simulation.

2. In the pseudocode procedure **Update W for Arrival**(P), in the assignment

$$l^* \leftarrow \arg \min_{l:g(l,d)>0} \; [\kappa W(l,d) + U(l,d) + \omega]/g(l,d) \; ,$$

ω was taken to be the packet size $|P|$ instead of 1 bit. This was changed beginning in the 15 August 1993 version of the simulation.

The last two differences between the STIP3 pseudocode and the STIP3 simulation code are minor bugs. These bugs are still present in the latest version of the STIP3 simulation (15 August 1993) delivered as of this writing. These bugs will be corrected in a future version of the STIP3 simulation to be released soon.

3. The pseudocode procedure **Update Private Distance Variables**() was executed only if *epochs_since_last_update* was greater than or equal to *short_update_period*, the same as for **Update Reported Distance Variables**(). This resulted in *e_priv*(d) being allowed to change only in those epochs in which $e(d)$ is allowed to change. This difference has no effect unless *short_update_period* > 1.

4. In the pseudocode procedure **Update Private Distance Variables**(), the "else if" condition for smoothing $e\_priv(d)$ when it decreases was $(e'(d) < e(d) < emax)$ instead of $(e'(d) < e\_priv(d) < emax)$.

8 Performance Analysis

In this section, we describe our analysis of the behavior of STIP3 under various topological and traffic conditions. In order to gauge the performance and behavior of STIP3, SRI constructed a simpler baseline protocol (also referred to as Baseline) to be subjected, in almost all cases, to the same tests. Baseline uses the same link computations and maintenance, expected delay smoothing and thresholding, and queue prioritization algorithms that are used with STIP3. However, Baseline uses simple forwarding and alternate routing rules in place of the sophisticated flow-computation and link-scheduling algorithms used in STIP3. Specifically, Baseline differs from STIP3 in the following ways:

- Acknowledgments (acks) and updates are transmitted on the same link as the data packet to which the acknowledgment applies. (STIP3 has the option to send acks and updates over indirect paths.)
- Traffic packets are always enqueued onto the best link's queue; thus, packets queue up before alternate paths are taken. (STIP3 has the option to select one of two flow computation algorithms; one minimizes a local objective [referred to as "old flows" and identical to the flow computation used for STIP2], and the other minimizes both a local objective function and a global objective function [referred to as "new flows"].)
- Packets are dropped only if there is no good link to the destination. (STIP3 has a time to live for both traffic and control packets.)
- Spreading, directed acyclical graphs (DAGs), and high-priority updates are not used or considered (unlike STIP3).

Our simulations were characterized by the number of nodes (10, 25, and 50) and by network dynamics (none, slow [2 s, 5 s, and 10 s jamming period], and fast [0.25 s jamming period]; random, spot, and circular jamming). For STIP3, default parameter settings are identical for all simulations except for gamma (gamma = 100 for no or slow jamming; gamma = 1 for fast jamming). It should be noted that for STIP3, sending acks/updates only over direct links, spreading of 0.2, and use of new flows are default conditions. Finally, it should also be noted that for STIP3, the

epoch interval is 0.2 s and for Baseline it is 2.0 s on the average. Default parameter values should be assumed in the discussion below, unless otherwise stated.

Below, we summarize our experiment results. these results presented below are based upon the 2 August version of the Baseline protocol and the 29 July version of STIP3, unless noted otherwise; and are a superset of the analysis presented at the final review meeting for EDMUNDS. We designed 22 different experiments and executed over 300 simulations. We begin by presenting the saturation point of the protocols where we gradually increased the offer rate of the traffic until saturation was reached; these tests were performed on simple (e.g., linear and fully connected) topologies. The effects of particular differences (e.g., flow computation, dynamic calculation of the smoothing constant for link probabilities) between STIP3 and Baseline are then shown using specifically designed experiments. We then compare the overall end-to-end (ETE) delay performance of both protocols under small, medium, and large networks and in scenarios with varying degrees of dynamics.

8.1 Effective Network Capacity

A number of different simulations were run on simple topologies with no dynamics, to determine the effective network capacity for a single source/destination pair, using STIP3 and Baseline.

Four different scenarios were used; in each case the number of outgoing links from the source to the destination was different. Unless otherwise stated, the capacity of each link was 10,000 bits per second (bps) and the size of the traffic packet was 943 bits; thus the link capacity is approximately 10 packets per second (pps). With each scenario, we gradually increased the packet offer rate at the source in increments of 1 pps until there was a significant decrease in expected end-to-end delay. The STIP3 old flows option was used for all these simulations. We noted that for simple topologies with low connectivity and little dynamics, old flows generally performed better than new flows.

Our results showed that STIP3 operates with less channel overhead and thus a higher maximum throughput (approximately 6% higher than Baseline with these experiments). However, for the same end-to-end delay, STIP3 achieves up to twice as much throughput as Baseline.

To assist in our analysis, we define the effective network capacity of a protocol in a particular scenario as the maximum rate of traffic flow for a single source/destination pair, divided by the sum of the raw capacities of the links in the minimum cut of the network topology between the source and destination. In general, the average effective network throughput ranged from 90% to 95% for STIP3 and from 85% to 92.5% for Baseline.

8.2 Behavior of STIP3 Features

In this section we briefly discuss the effects of key features of STIP3 in comparison with corresponding features of Baseline. This discussion provides some detailed analysis of the features of STIP3, which will be useful in understanding the basic end-to-end delay performance comparison between STIP3 and Baseline in Subsection 8.3.

Flow Computation. The Baseline protocol sends traffic on the current best link and builds up its link queue before taking an alternate path. For example, for a constant stream that uses two outgoing links, this algorithm causes a permanent queuing delay for the primary link; this queuing delay is approximately equal to the ETE delay for packets using the secondary link. In STIP3, both old flows and new flows overcome this problem.

For old flows, the objective function used to compute the flows over each link for each destination considers the delay of packets that are expected to arrive at the node in the near future. Rather than minimize this local objective function, the new flow computation uses marginal cost to minimize a global function.

The effects of these differences are evident when a simple topology ("3-path") is used. In this topology, there are three paths from the source to the destination; one path is one hop from the source to the destination, and each of the other two paths is five hops from the source to the destination. In this example, when an offer rate is used that requires at least two outgoing links to be used at the source, the mean ETE delay with STIP3 is approximately 42% less than that for Baseline. The reason for this difference is that Baseline has a ~0.5 s permanent queuing delay on the primary link (i.e., approximately equal to the ETE delay for traffic taking the longer paths) whereas STIP3 has a queuing delay of only ~0.2 s on the primary link.

In this simulation, we noted oscillations in the STIP3 mean ETE delay for packets taking the one-hop path. This can be explained by the fact that in the 29 July version of STIP3, the local flow computation did not always converge to an optimal solution. Since this version of STIP3, we have made a few minor optimizations, which are included in the 15 August version of STIP3. In particular, the 29 July version was enhanced to update the average expected delay to the destination (i.e., eavg) whenever it is greater than 1/s3_phi where s3_phi is the decay rate for future time averaging of the queuing delay. With the new version of STIP3 we used a smaller step size (s3_step = 0.1 rather than the default value, 1) and a larger number of steps (s3_num_steps = 300 rather than the default value, 30) for gradient descent, and placed more emphasis on the number of queued packets assigned to a link (by setting s3_W_coeff = 100 rather than the default value, 1). (s3_num_steps = 300 rather than the default value, 30) Our results showed further improvement of STIP3 over Baseline: the mean ETE delay with the optimized STIP3 is approximately 45% less than that for Baseline.

Directed Acyclical Graphs. If the primary link fails, Baseline immediately forwards traffic on a non-DAG alternate outgoing link, without knowing if the alter-

nate link goes to the destination or not, unlike STIP3, which sends traffic only along a DAG.

The usefulness of DAGs can dramatically be seen in a simple experiment in which the "dogbone" topology was used with two streams (S1, S2) of traffic (each from one end of the dogbone to the other, and with different sources and destinations), and the end link of stream S2 was simply brought down. When the link was brought down, the Baseline protocol began to forward S2 traffic upstream, adversely affecting S1; in this example, Baseline realized after ~20 s that the S2 traffic cannot reach its destination, and begins to drop S2 traffic packets. On the other hand, STIP3 realizes almost immediately that traffic cannot reach the destination and queues packets at the source without effecting S1. Results showed that the mean ETE delay for S1 with STIP3 is approximately 60% less than that for Baseline. It should be noted that old flows was used in this experiment

Link Scheduling. In our analysis of STIP2, which used a leaky-bucket control mechanism, we noted that STIP2 did not recognize increases in traffic rates due to a burst, and gave preferential treatment to constant streams. To correct this problem, STIP3 uses an FCFS algorithm to service its links – traffic on a given link is transmitted in the order of its time of arrival regardless of its destination.

The improvement of STIP3 over STIP2 was obvious when the "dogbone" topology was used, with two traffic streams (one continuous and the other bursty, with 1-4 s interburst intervals and burst lengths). The increase in mean ETE delays from the constant stream to the bursty stream is ~16% to ~98% with STIP2 and ~5% to ~17% with STIP3; the longer the interburst interval use in the simulation, the greater the increase in mean ETE delays. With Baseline, the increase in mean ETE delay ranged from ~5% to ~24%; STIP3 is slightly more fair than Baseline to bursty traffic as interburst lengths increase. The old flows method was used in this experiment.

Spreading. Our previous performance analysis of STIP2 described the benefits of spreading. For completeness, we reiterate the effects of this feature with an example from a STIP3 simulation using old flows. We used a simple 25-node ring topology. The scenario was such that there were two paths from the source to the destination – one with 11 hops, the other with 14 hops; the link capacities used were the default except that the tenth link from the source on the shorter path has a capacity of 5,000 bps. At a rate of 8 pps, traffic can be sustained either on the longer path alone, or on both paths. With the Baseline, traffic switches between the two paths; traffic is sent along one path until the protocol determines that the other path is better. On the other hand, STIP3 "spreads" traffic onto both streams. Compared to Baseline, STIP3 resulted in a decrease in mean ETE delay of 31% using a spreading value of .2, and 36% using spreading of 0.5; spreading helped reduce oscillations. Based upon a number of other simulations, our optimal value for spreading is approximately 0.2.

Priority Scheduling. STIP3 uses a priority scheduling algorithm allowing for limited precedence to higher-priority packets without necessarily excluding packets with lower priority. Two experiments were designed and executed, using a simple

topology with two traffic streams. Each stream had a different priority and each was parameterized by specific leaky-bucket constraints. The paths for both streams were nondisjoint, except for the last hop. As expected, our tests demonstrated that lower-priority traffic was either squeezed out or maintained at a specified level after network capacity was reached. Thus, the priority scheduling option can be used, for instance, to limit control traffic so that it does not utilize more than a certain percentage of the link capacity.

In addition, an experiment was designed to test priority scheduling, using a random network with two streams of traffic taking somewhat disjointed paths. In this particular case, the results were not quite as expected. STIP3 priority scheduling is performed on a link-wise basis and does not perform well for global priorities. To help resolve this problem, and as noted in the conclusions below, flow computation should consider priority restrictions.

Dynamic Calculation of Gamma. Both Baseline and STIP3 use a parameter called gamma which is the smoothing parameter used to update the smoothed link probability. During our analysis of STIP1 and STIP2, we noted that the optimal value of gamma depends upon network dynamics. In particular, our tests confirmed our expectations that with slow dynamics it is better to use less smoothing of link probabilities; and with fast dynamics it is better to use more smoothing, so that the protocol does not react quickly to these network changes. Our optimal choices for gamma are 100 for static scenarios and scenarios with slow dynamics, and 1 for scenarios with fast dynamics. With STIP3, there is an option to automatically set the value of gamma depending upon link changes, in contrast to Baseline, STIP1, and STIP2, where gamma is set once at the beginning of the simulation.

As expected, the performance of STIP3 with automatic setting of gamma is better than that of STIP3 and Baseline with preset values for gamma (i.e., 1 or 100) in scenarios where the rate of link dynamics changes during the simulation between slow and fast (e.g., between 5 s and 0.25 s jamming periods). In a simple random 10-node network topology where the speed of a jammer changes every 10 s (alternating between 0.25 s and 2 s), the mean ETE delay for STIP3 with automatic setting of gamma was lower by ~20-25% compared to Baseline, and lower by ~10-15% compared to STIP3 without dynamic setting of gamma.

8.3 End-to-End Delay Performance

In this section we discuss the overall performance of STIP3 and Baseline in scenarios of 10, 25, and 50 nodes, with and without dynamics, and using random topologies. A number of experiments using 10-node simple topologies were also performed (e.g., ring or "3-path").

Small Static Networks. In small, simple networks (10 nodes) with no dynamics, STIP3 generally outperforms Baseline. Our results show STIP3's mean ETE delay is up to 45% lower than Baseline's. In more complex, random networks, no

definitive conclusions can be made; in some cases STIP3 performance was better (8% decrease in mean ETE delay) and at other times it was worse (14% increase in mean ETE delay). In addition, in one experiment, Baseline's mean ETE delays were lower than STIP3's, but at the expense of lower reliability. The overall decrease in the performance in more complex, randomly connected networks may in part be explained by the increase in the number of control packets and increased number of traffic streams.

Medium-Size and Large Static Networks. In medium-size and large, random networks (25-50 nodes) with no dynamics, STIP3 performs better than Baseline. With 25-node networks, mean ETE delays were ~6% to 25% lower with STIP3 than with Baseline. In the only simulation performed on 50-node networks with no dynamics, the mean ETE delays were up to 11% lower with STIP3 than with Baseline; if high reliability is important, STIP3 outperformed Baseline in that the reliability of one of the three traffic streams for Baseline was very low (~33%). Additional simulations need to be performed.

Small and Medium-Size Dynamic Networks. A number of simulations with both slow (5 s and 10 s jamming periods) and fast (0.25 s jamming periods) dynamics were run with small and medium-size networks (10-25 nodes). In general, our experiments showed that with slow dynamics, STIP3 outperformed Baseline. The mean ETE delays for Baseline were approximately 1.0 to 12.5 times greater in slow-dynamics scenarios than in scenarios with no dynamics. With STIP3, the mean ETE delays were only about 1.0 to 4.0 times greater in the slow-dynamics scenarios than in scenarios with no dynamics. Additionally, our results showed that STIP3 mean ETE delays were about 15-85% lower than those for Baseline in scenarios with slow dynamics. It should be noted that with random jamming of 10-node networks, we found that indirect routing of acks and updates should be used (mean ETE delay values were over two times higher when we used direct routing of acks and updates).

With fast dynamics, STIP3 at times performed better than Baseline (showing a decrease of up to 66% in mean ETE delays) and at times worse (with up to 32% increase in ETE delays).

Large Dynamic Networks. We performed one experiment using a 50-node network with both slow and fast dynamics. The reliability of both Baseline and STIP3 was often low. We suspect that in such large networks, there are a large number of packet retransmissions due to failed acknowledgments; these retransmissions cause high ETE delays and therefore low reliability.

With fast jamming, STIP3 outperformed Baseline; mean ETE delays were 60-75% lower and reliability was ~75-200% higher compared to Baseline. With slow jamming, it is a little more difficult to draw definitive conclusions. In our particular example, STIP3 mean ETE delays were 30% to 47% lower than Baseline; however the reliability of one of the three streams was 77% for Baseline, but only 37% for STIP3. Depending upon the traffic requirements (e.g., high reliability vs. low ETE delay), STIP3 may be better or worse than Baseline.

Responsiveness. In order to demonstrate the responsiveness of the protocols to a link change, we used a simple 25-node ring topology where one path (P1) between the source and destination was shorter (12 hops) than the other path (P2: 13 hops). Traffic was offered at a rate below the link capacity so that traffic could be sustained on only one path; thus, traffic would initially take P1. A series of simulations were performed, with each simulation removing a link one further down path P1 towards the destination than the previous simulation. The recovery times (i.e., the time from the link breakage to the time when traffic started on path P2) were examined. In each instance, STIP3's recovery time was smaller than Baseline's; and, as expected, the recovery time generally increased as the link removed was farther from the source. Recovery times for STIP3 ranged from near 0 to ~4.25 s; and ~1 to 16.5 s for Baseline. Additionally, results showed that the recovery times for Baseline are up to 3.7 times higher than those for STIP3. This is likely to be due in part to the difference in the epoch interval between the two protocols, and to STIP3's use of high-priority update packets to advertise changes that affect one or more of the destination-oriented DAGs.

Fairness. Both STIP3 and the Baseline protocols are fair. For example, experiments with multiple streams of traffic from different sources to the same destination over the identical path (except the first link), resulted in ETE delays which were statistically the same.

Topologies with Multimedia Links. Link capacities in the experiments described above were uniform (i.e., 10,000 bps). In a scenario with links of various capacities (i.e., 5,000, 1,0000, and 15,000 bps), we observed that STIP3 handles networks with different link capacities better than Baseline. On the average, STIP3's mean ETE delays were ~35% lower than Baseline. Additional experiments with multimedia links need to be performed.

8.4 Conclusion

STIP3 contains a number of improvements over STIP2 and STIP1. These include new link- state estimation, link scheduling, and flow computation algorithms. As could be expected however, our simulations and analyses identified deficiencies in certain algorithm components of STIP3. These deficiencies are discussed below. Nevertheless, simulations showed that STIP3 performs better than Baseline in most experiments, and in many cases, is far superior.

The following lists our main findings from simulations run on STIP3 and on Baseline.

- For the same end-to-end delay, STIP3 achieves up to twice as much throughput as Baseline.
- STIP3 and Baseline are fair under bursty traffic conditions.

- Baseline was found to have as much as 6.7 times the mean end-to-end delay and as much as 3.7 times the response time of STIP3.
- For STIP3, old flows generally performed better than new flows in simple topologies with low connectivity and little dynamics.
- For STIP3, acknowledgments and updates should be sent over indirect links in scenarios with slow, random jamming in small networks (10 nodes).
- STIP3 generally outperforms Baseline in small, simple networks with no dynamics (up to 45% lower mean ETE delays).
- STIP3 performs better than Baseline in medium-size and large random networks with no dynamics (5-25
- STIP3 outperforms Baseline in small and medium-size networks with slow dynamics (15-85% lower ETE delays).
- STIP3 outperforms Baseline in large networks with fast dynamics (60-75% lower ETE delays; 75-200% higher reliability).
- Baseline's reliability is at times much lower than STIP3's, especially with large-sized networks with "heavy" traffic or high dynamics.

  item Additional tests need to be performed before definitive conclusions can be reached regarding STIP3 and Baseline performance in small random networks with no dynamics, small and medium-size networks with fast dynamics, and large networks with slow dynamics.

8.5 Improvements for STIP3

As a result of our simulations, we have identified a number of improvements to STIP3 (27 July version), which are mentioned above in our discussions that focus on the STIP3 flow computation algorithm. These improvements have been implemented into the 15 August version of STIP3. In addition, we have identified a number of potential improvements to this latest version of STIP3. These include the following:

- Flow computation should depend upon priority restrictions.
- The number of redundant packet replications due to failed acks should be reduced.
- The convergence of the gradient descent method should be improved.

Bibliography

[1] D.P. Bertsekas and J.N. Tsitsiklis. *Parallel and Distributed Computation.* Prentice-Hall, Inc., 1989.

[2] D.G. Luenberger. *Introduction to Linear and Nonlinear Programming.* Addison-Wesley, 1973.

Appendix F

PAPERS PUBLISHED UNDER THE EDMUNDS PROJECT

MINIMUM-EXPECTED DELAY ALTERNATE ROUTING *

Richard G. Ogier and Vlad Rutenburg

SRI International
Menlo Park, CA 94025

Abstract

This paper considers the problem of finding a routing strategy that minimizes the expected delay from every source to a single destination in a network in which each link fails and recovers according to a Markov chain. We assume that each node knows the current state of its own outgoing links and the state-transition probabilities for every link of the network. We show that the general problem is #P-complete and consider two special cases: Case 1 assumes the network is a directed acyclic graph (DAG) oriented toward the destination, and Case 2 assumes that the link states are independent from slot to slot. For each case, we prove that the optimal routing strategy has a simple state-independent representation and we present an efficient algorithm for finding the optimal strategy.

1 Introduction

We consider the problem of minimum-expected-delay routing in a highly dynamic, unreliable network. We assume that the network has an underlying topology given by a directed graph $G = (V, E)$, where E represents the set of links that are potentially operational. Each link in E switches according to a Markov chain between two possible states: up and down. The Markov chains for different links are independent. To simplify the presentation, we assume that time is divided into slots of equal length. We let $\lambda(i,j)$ denote the probability that link (i,j) will come up in the next slot given that it is down in the current slot, and we let $\mu(i,j)$ denote the probability that link (i,j) will go down in the next slot given that it is up in the current slot. It follows that the steady-state probability that link (i,j) is up is

$$p(i,j) = \frac{\lambda(i,j)}{\lambda(i,j) + \mu(i,j)} \quad . \tag{1}$$

*This work was supported by the Defense Advanced Research Projects Agency and the Rome Air Development Center under Contract F30602-90-C0003, and by the U.S. Army Research Office under Contract DAAL03-88-K-0054.

(If $\lambda(i,j)$ and $\mu(i,j)$ are both zero, then $p(i,j)$ can be any number between 0 and 1.) Link (i,j) has a delay of $d(i,j)$ slots when it is up, which can represent propagation and queuing delays.

We assume that G, $d(i,j)$, $\lambda(i,j)$, $\mu(i,j)$, and $p(i,j)$, for all links (i,j), are static and known by all nodes. (In practice they could be updated periodically.) We also assume that each node i knows the current state of its own outgoing links (i,j); all other links (k,l) are assumed to have the steady-state probability $p(k,l)$ of being up. Let z be a given destination node. We consider the following *minimum-expected-delay routing problem*: find a routing strategy that minimizes, over all possible routing strategies, the expected time for a packet originating from any node to reach z.

This problem is not equivalent to finding a shortest path from each source to z, since the links of any single path can change their state while the packet is en route. Instead, the routing decision at each node of the packet's journey must depend on the current state of the node's outgoing links. This problem is also not equivalent to finding k shortest paths with distinct initial links from each source to z [1]. Although the solution to the latter problem provides alternate paths that can be used in case an initial link fails, it does not consider the existence of alternate paths that can be taken in case a noninitial link of the path fails, potentially resulting in a much higher delay. This is illustrated in Figure 1, where we assume all links have identical parameters with $p(i,j) < 1$. The k-shortest path solution could result in a packet taking the link on the left, whereas the minimum-expected-delay solution would take one of the links on the right since they lead to nodes with more alternate routes.

We emphasize that the problem we consider is to find the minimum-expected-delay routing strategy among *all* possible strategies, including those that are history dependent (in which the decision at each hop depends on where the packet has traveled so far) and those that contain loops. Figure 2 gives an example for which the optimal routing strategy contains a loop. In this example, $d(i,j) = 1$ for all links, links $(1,2)$ and $(2,1)$ are always up, $p(1,z) = p(2,z) = .5$, and $\lambda(1,z) = \lambda(2,z) = .1$. Thus, if link $(1,z)$ or

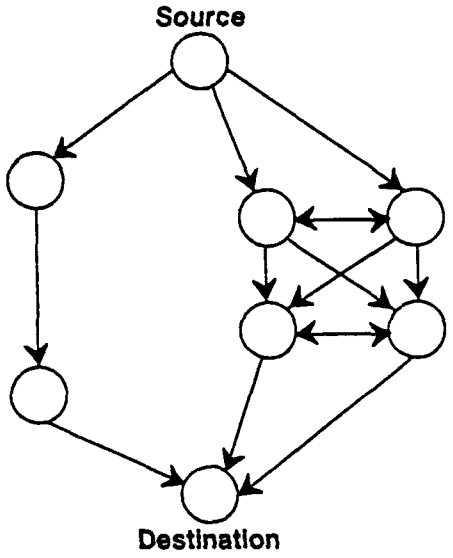


Figure 1: An example showing the advantage of our method.

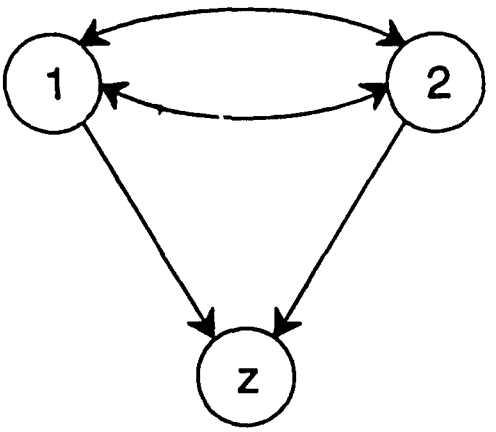


Figure 2: A network in which the optimal routing strategy contains a loop.

$(2, z)$ is down, it will stay down for an average of 10 slots. Clearly, if both of these links are down, then the minimum-expected-delay routing strategy is for a packet to hop between nodes 1 and 2 until one of the down links comes up.

The general minimum-expected-delay routing problem as stated is very difficult. In fact, we show in the next section that it is #P-complete (the enumeration version of NP-complete) by transformation from the two-terminal reliability problem, which was shown by Valiant [2] to be #P-complete. We therefore cannot expect to solve the general problem in polynomial time, and so we consider the following two more tractable cases:

Case 1 G is a DAG (directed acyclic graph) oriented toward z. Equivalently, G contains no cycles and z is the only node with no outgoing link.

Case 2 The states of each link are independent from slot to slot. Equivalently, $\lambda(i,j) = p(i,j)$ for all links.

For these two cases, we present centralized algorithms with surprisingly low time complexities. For Case 1, the time complexity is $O(|E| \log \Delta)$, where Δ is the maximum number of outgoing links for any node. For Case 2, the time complexity is $O(|E|[\Delta + \log_{(1+|E|/|V|)} |V|])$. If we assume unit link delays, this complexity can be reduced to $O(|E| \log_{(1+|E|/|V|)} |V|)$, which is the same as for Dijkstra's shortest-path algorithm.

In addition, we show that the optimal routing strategy for either case has the following form as shown in Figure 3: Each node i is assigned an ordering on its outgoing links (best to worst) and an integer $K(i)$. We let P_i denote the ordered set containing the $K(i)$ best links at node i. The links of P_i are called *preferred links*. If node i has a packet to forward, and at least one preferred link is up, then the packet is forwarded on the best such link; otherwise node i waits for a preferred link to come up. P_i does not depend on the current state of node i's outgoing links (it only depends on the quasi-static information G, d, λ, and μ).

Since the set P_i provides alternate links that are computed in advance of any link failures, the optimal solution takes the form of alternate routing. We therefore call our solution *minimum-expected-delay alternate routing* (MEDAR). We emphasize that, although P_i does not depend on the the current states of node i's outgoing links, the MEDAR solution is optimal among all routing strategies that do depend on these states. The proof of this uses the theory of Markov decision chains.

MEDAR provides the following practical benefits:

- It makes optimal use of statistics for link-state dynamics to minimize expected delay.
- By computing a quasi-static routing structure that provides alternate paths, it requires less frequent topology updates and avoids the delay of computing a new path if a link fails.
- It allows the option of waiting for an outgoing link to come up rather than immediately using an inferior link.

The rest of the paper is organized as follows. In Section 2 we show that the minimum-expected-delay routing problem is #P-complete. In Section 3 we solve the

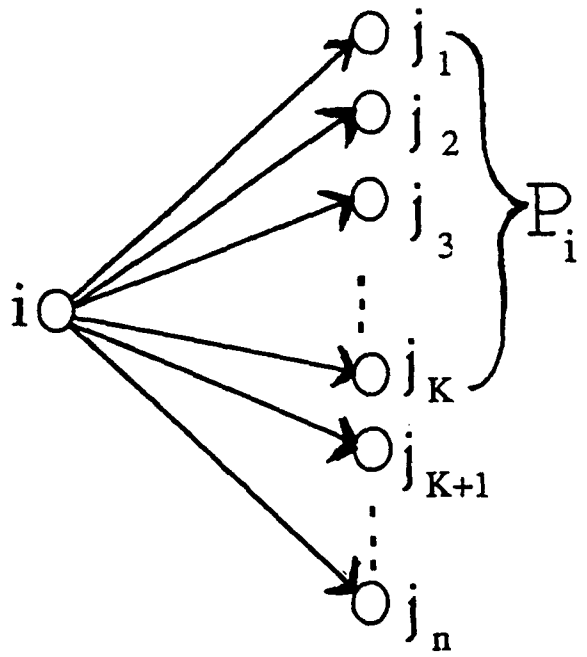


Figure 3: Optimal routing strategy for node i.

local routing problem that assumes each node knows the expected delay from each neighbor to z. In Section 4 we show how the local solution can be applied to obtain efficient algorithms for Cases 1 and 2 of the minimum-expected-delay routing problem. Finally, Section 5 discusses heuristics for the general problem.

2 #P-Completeness

Proposition 1 *The minimum-expected-delay routing problem is #P-complete.*

Proof We prove the proposition by showing that the two-terminal reliability problem can be reduced to the minimum-expected-delay routing problem. Given a directed graph G' in which each link has a probability $p(i,j)$ of being up, the two-terminal reliability problem is to find the probability that there exists a path from a source s to a destination z consisting of links that are up. Given an instance of this problem, we construct an instance of the minimum-expected-delay routing problem as follows. We let G be G' with an additional node k and additional links (s,k) and (k,z). We let $p(i,j)$ be the same as for the reliability problem for links in G', and we set $p(s,k) = p(k,z) = 1$. We let $\lambda(i,j)$ and $\mu(i,j)$ be zero for all links, so that links never change state. Finally, we let $d(i,j) = 0$ for all links of G', and we set $d(s,k) = d(k,z) = .5$. The minimum-expected-delay routing strategy from s to z is clearly to first try to find a path (of zero delay) in G', and, if this fails, to use the path $(s,k)(k,z)$ of delay 1. The minimum expected delay is therefore $1 - r$, where r is the probability that a path exists in G'. Therefore, finding the minimum expected delay also gives the solution to the two-terminal reliability problem, proving our claim. ■

3 Solution to the Local Routing Problem

In this section, we consider a fixed node i, and we assume that node i knows the expected delay from each neighbor to the destination z. We will determine the local strategy for node i that minimizes the expected delay for a packet to reach the destination. Throughout the paper, we assume steady state, so that the probability that each link (j,k) is up is $p(j,k)$.

For each outgoing link (i,j) of node i, we let e_j denote the expected time for a packet to reach z from node j for some (not necessarily optimal) fixed routing strategy, and we let $e(i,j) = d(i,j) + e_j$ denote the expected time to reach z from node i assuming that link (i,j) is up and is selected. We assume in this section that the time for a packet to reach z from each neighbor j (which is a random variable) is independent of the state of the outgoing links at node i, and of the strategy used by node i. This assumption holds, for example, if the packet is guaranteed never to return to node i. It follows that $e(i,j)$ is independent of the strategy used by node i, and is not changed if we condition the expectation on the states of the outgoing links at node i.

The problem of this section is actually more general than the local routing problem, since choosing link (i,j) can represent any action, $e(i,j)$ can be any number representing the cost of that action, and the Markov chain for link (i,j) determines when that action can be taken.

Let us restate our objective. When a packet arrives at node i, each outgoing link is in one of two states: up or down. Letting Δ_i denote the number of outgoing links at node i, there are 2^{Δ_i} possible configurations of links. For each such configuration, we need to specify what action to take: whether to wait or not, and in the latter case, which up link to take. Thus, the task of computing these 2^{Δ_i} actions could be exponential. We will show to the contrary that the optimal strategy is very simple and requires only $O(\Delta_i \log \Delta_i)$ time to compute.

We assume that it is not possible for all outgoing links to be down for all time. That is, we assume that at least one outgoing link has a nonzero probability $\lambda(i,j)$ of coming up when it is down.

We start with an observation that simplifies the problem. Since node i knows which of its outgoing links are currently up, the optimal policy at each time slot is

clearly either to choose the link of minimum $e(i,j)$ that is currently up or to wait for the next time slot. It follows that the problem is a special case of the optimal stopping problem, which in turn is a special case of a Markov decision chain [3].

Let (i,j_k) denote the kth outgoing link at node i, numbered in order of increasing delay to the destination, i.e., $e(i,j_k) \leq e(i,j_{k+1})$. By "the best k links" we will mean the set $\{(i,j_1),...,(i,j_k)\}$. For a given k, let ρ_k denote the following policy: If at least one of the best k links is up, then use the best such link; otherwise wait for such a link to come up. At the end of this section we will show that the optimal policy has the form ρ_k for some k. We first restrict our attention to policies of this form, and we present an algorithm for computing the optimal value of k.

Assuming we are using the policy ρ_k for some given k, the resulting expected delay, denoted $e_i(k)$, satisfies the dynamic programming equation

$$e_i(k) = \sum_{l=1}^{k}[\prod_{m=1}^{l-1}(1-p(i,j_m))]p(i,j_l)e(i,j_l) + [\prod_{l=1}^{k}(1-p(i,j_l))]f_i(k) \quad , \tag{2}$$

where $f_i(k)$ denotes the expected delay to the destination given that the best k links are currently down. The lth term of the sum is equal to the probability that the lth best link is the best link that is up, times the expected delay assuming that the packet is sent on this link. (For $l = 1$, the product within the sum is assumed to be unity.) The last term is equal to the probability that the best k links are all down, times $f_i(k)$.

The number $f_i(k)$ satisfies the recursive equation

$$f_i(k) = 1 + \sum_{l=1}^{k}\prod_{m=1}^{l-1}[(1-\lambda(i,j_m))]\lambda(i,j_l)e(i,j_l) + [\prod_{l=1}^{k}(1-\lambda(i,j_l))]f_i(k) \quad , \tag{3}$$

where the first term expresses the fact that if the first k links are currently down, then we must wait at least one time slot. Recall that $\lambda(i,j)$ is the conditional probability that link (i,j) will come up in the next slot given that it is currently down. The last term expresses the fact that if the first k links are still all down in the next time slot, then the expected delay will still be $f_i(k)$.

We now consider the problem of finding the optimal value of k. Solving the recursive equation (3) for $f_i(k)$, we obtain

$$f_i(k) = \frac{R_k}{1-L_k} \quad , \tag{4}$$

where

$$R_k = 1 + \sum_{l=1}^{k}[\prod_{m=1}^{l-1}(1-\lambda(i,j_m))]\lambda(i,j_l)e(i,j_l) \tag{5}$$

$$L_k = \prod_{l=1}^{k}(1-\lambda(i,j_l)) \quad . \tag{6}$$

These can be computed for $k = 1,2,...$ using the following recursion:

$$R_{k+1} = R_k + L_k\lambda(i,j_{k+1})e(i,j_{k+1}) \quad , \tag{7}$$

$$L_{k+1} = L_k(1-\lambda(i,j_{k+1})) \quad , \tag{8}$$

where $R_0 = L_0 = 1$. This recursion will be used below in an efficient algorithm for computing the optimal k. We first need the following theorem, which characterizes the optimal k.

Theorem 1 *Let K be the largest integer such that*

$$f_i(k-1) > e(i,j_k) \text{ for all } k \leq K \quad , \tag{9}$$

where we take $f_i(0)$ to be ∞. Then K minimizes $e_i(k)$ over all k. Even stronger, for any initial state of the outgoing links, the policy ρ_K results in an expected delay no greater than any policy ρ_k for $k \neq K$.

This theorem immediately suggests the following algorithm for computing K: Compute $f_i(k)$ for $k = 1,2,...$ until either $f_i(k) \leq e(i,j_{k+1})$ or k is equal to the number of outgoing links. Then set $K = k$. Rather than using Equations (5) and (6) to compute R_k and L_k for each k, a more efficient method is to iterate Equations (7) and (8) to compute R_k and L_k for $k = 1,2,....$ Each such iteration requires constant time, and at most Δ_i iterations are required, where Δ_i is the number of outgoing links. Therefore, the running time of the algorithm for computing K is $O(\Delta_i)$, assuming that the outgoing links are already sorted in order of increasing delay. This sorting requires $O(\Delta_i \log \Delta_i)$ time, which is therefore the total running time of the algorithm.

The proof of the theorem will use the following lemma.

Lemma 1 *If $f_i(k) > e(i,j_{k+1})$, then $f_i(k) > f_i(k+1)$. It follows that if K satisfies the condition of the theorem, then $f_i(1) > f_i(2) > \cdots > f_i(K)$.*

Proof Suppose $f_i(k) > e(i,j_{k+1})$. Substituting Equations (7) and (8) into Equation (4) gives

$$f_i(k+1) = \frac{R_k + L_k\lambda(i,j_{k+1})e(i,j_{k+1})}{1 - L_k(1-\lambda(i,j_{k+1}))} \tag{10}$$

$$< \frac{R_k + L_k\lambda(i,j_{k+1})\frac{R_k}{1-L_k}}{1 - L_k(1-\lambda(i,j_{k+1}))} \tag{11}$$

$$= \frac{R_k}{1-L_k} \tag{12}$$

$$= f_i(k) \quad . \tag{13}$$

where (11) follows from the assumption $f_i(k) > e(i, j_{k+1})$ and Equation (4), (12) follows from routine manipulations, and (13) follows from (4). ■

Proof of Theorem 1 Let K satisfy the condition of the theorem. We will prove the stronger result that for any initial state of the outgoing links, the policy ρ_K results in an expected delay no greater than any policy ρ_k for $k \neq K$. We consider two cases.

First suppose that the best K links are all down. Then using policy ρ_K results in an expected delay of $f_i(K)$. Similarly, using policy ρ_k for any $k < K$ results in an expected delay of $f_i(k)$, which by the lemma is greater than $f_i(K)$. Consider the policy ρ_k for any $k > K$. Suppose one of the best k links (excluding the best K links) is currently up, and that the best such link is link l. Then this policy results in a delay of $e(i, j_l)$, which is greater than or equal to $f_i(K)$ since the condition of the theorem implies $f_i(K) \leq e(i, j_{K+1}) \leq e(i, j_l)$. Otherwise, let link l be the best link that first comes up among the k best links. If $l \leq K$, then both policies ρ_k and ρ_K would choose this link and would thus result in the same delay. If $l > K$, then policy ρ_k would still choose link l, resulting in a delay $e(i, j_l)$ plus the time to wait for link l to come up, which as before is greater than $f_i(K)$.

Next suppose that at least one of the best K links is up, and that the best such link is link l. Then policy ρ_K chooses link l and results in a delay of $e(i, j_l)$. Similarly, any policy ρ_k for $k \geq l$ results in the same delay. Finally, consider the policy ρ_k for any $k < l$. This policy results in an expected delay of $f_i(k)$, which by the lemma is greater than or equal to $f_i(l-1)$, which by the condition of the theorem (since $l - 1 < K$) is greater than $e(i, j_l)$, which is the delay resulting from ρ_K. Since all cases have been considered, the theorem is proved. ■

We end this section by proving that ρ_K is not only optimal over all policies of the form ρ_k, but is also optimal over all policies. For this we need new terminology and notation. A decision rule specifies what action to take as a function of the current state. A policy is a sequence of decision rules. If ϕ and ψ are decision rules, then ϕ^∞ denotes the stationary policy that uses ϕ at every step (time slot), and $\psi\phi^\infty$ denotes the policy that uses ψ at the initial step and ϕ thereafter. Let ϕ_k denote the following decision rule: If at least one of the best k links is up, then use the best such link; otherwise wait. Then the policy ρ_k defined above is equal to ϕ_k^∞.

As stated above, we assume that at least one link has a nonzero probability of coming up when it is down. We let l_0 denote the best such link. We define a policy or decision rule to be *admissible* if it chooses not to wait whenever link l_0 is up. Since no link better than l_0 will ever come up if it is down, every optimal policy must be admissible, and thus we can restrict our attention to admissible policies and decision rules. This restriction guarantees that every stationary admissible policy is *transient*, i.e., eventually chooses a link. It follows [3, page 140] that there exists a stationary optimal policy.

Let $J(\sigma)(x)$ denote the expected cost (delay) of using policy σ when the initial state is x. To prove that ρ_K is optimal, we will use the Comparison Lemma [3, page 137], which for our purposes can be paraphrased as follows:

Comparison Lemma *Suppose ϕ^∞ and ψ^∞ are transient policies. Then the following three statements are equivalent:*

$$J(\phi^\infty)(x) \leq J(\psi^\infty)(x) \text{ for all } x \quad (14)$$

$$J(\phi^\infty)(x) \leq J(\psi\phi^\infty)(x) \text{ for all } x \quad (15)$$

$$J(\phi\psi^\infty)(x) \leq J(\psi^\infty)(x) \text{ for all } x \quad (16)$$

Theorem 2 *Strategy ρ_K is optimal, i.e., the best among all strategies. Namely, for each initial state x of the outgoing links, $J(\rho_K)(x) \leq J(\sigma)(x)$ for all policies σ.*

Proof To prove the theorem, it suffices to show that $J(\phi_K^\infty)(x) \leq J(\psi^\infty)(x)$ for all admissible decision rules ψ, since we know there exists a stationary optimal policy. (Recall that $\rho_K = \phi_K^\infty$.) Clearly, we only need to consider decision rules ψ such that, for any state x, ψ either chooses the best link that is up or waits. Let ψ be such a decision rule. By Theorem 1, $J(\phi_K^\infty)(x) \leq J(\phi_k^\infty)(x)$ for all x and all k, which by the Comparison Lemma implies $J(\phi_K^\infty)(x) \leq J(\phi_k\phi_K^\infty)(x)$ for all x and all k. Now for any state x there exists a k such that ϕ_k performs exactly the same action as ψ. (If ψ chooses the best link that is up, let k be greater than the number of this link; otherwise let k be less than the number of this link.) Therefore, $J(\phi_K^\infty)(x) \leq J(\psi\phi_K^\infty)(x)$ for all x, which by the Comparison Lemma implies $J(\phi_K^\infty)(x) \leq J(\psi^\infty)(x)$ for all x, which proves the theorem. ■

4 Algorithms for the Global Problem

In this section, we apply the solution to the local problem to solve the minimum-expected-delay routing problem for Cases 1 and 2 defined in Section 1. Recall that the solution to the local problem assumed that the time for a packet to reach z from any neighbor j is independent of the current state of the outgoing links at node i. This assumption does not hold for the general

problem, since, as shown in Section 1, even an optimal strategy can contain loops. However, this assumption clearly holds for Case 1, in which a packet can never return to the same node, and for Case 2, in which link states are independent from slot to slot. The solution to the local problem also assumed that the time for a packet to reach z from any neighbor j is independent of the policy at node i. This assumption clearly holds for Case 1, but it is not clear that it holds for Case 2, since an optimal solution for Case 2 can contain loops. However, we will show for Case 2 that there *exists* an optimal strategy with no loops. This fact will allow us to compute such a strategy by applying the algorithm for the local problem once to each node.

4.1 Case 1

In this case, we assume that the graph G is a DAG oriented toward the destination z (see Section 1). If the graph G is not a DAG, Section 5 discusses how to select a subgraph of G that is a DAG. For any such DAG, there exists an assignment of numbers $v(i)$ to the nodes such that if (i,j) is in the DAG, then $v(i) > v(j)$. An example is given in Figure 4. (In fact, the DAG can be defined in terms of these numbers, which can be obtained, for example, using a shortest-path algorithm.) The algorithm for this case is simply to apply the local algorithm of Section 3 to each node in order of increasing $v(i)$. Since a DAG contains no loops, the assumptions of Section 3 hold, and so this algorithm computes an optimal routing strategy. Since the running time of the local algorithm is $O(\Delta_i \log \Delta_i)$, where Δ_i is the number of outgoing links at node i, the total running time of the routing algorithm is $O(|E| \log \Delta)$, where $|E|$ is the number of links in the DAG and Δ is the maximum number of outgoing links for any node.

4.2 Case 2

In this case, we assume that the graph G is arbitrary but that the state of each link is independent from slot to slot, i.e., $p(i,j) = \lambda(i,j)$. In addition, we will assume that the link delays satisfy $d(i,j) \geq 1$, i.e., that it takes at least one time slot to send a packet over a link.

Given a routing strategy σ, let e_i^σ denote the value of the expected delay from node i to the destination under strategy σ. Let e_i^* denote the minimum (optimal) value of e_i^σ among all strategies σ. If $e_i^* = \infty$, then node i is of no interest to us, since this fact and the slotwise independence imply that there is and will never be a path from i to the destination. Thus, we eliminate all such nodes i.

The lemmas presented below allow us to efficiently compute an optimal routing strategy by processing one node at a time in order of increasing e_i^*. This procedure is similar to Dijkstra's shortest-path algorithm [4].

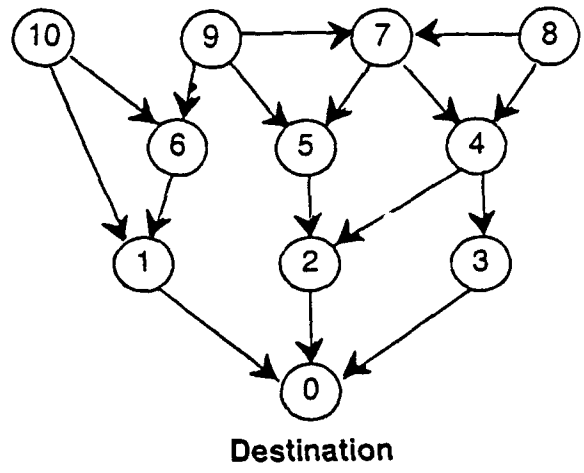


Figure 4: A destination-oriented DAG.

Our algorithm is as follows: A set S of nodes is maintained; initially S is empty. For each node i, a variable $e_i(S)$ is maintained which is the minimum expected delay using only intermediate nodes in S. Initially $e_z(S) = 0$ and $e_i(S) = \infty$ for $i \neq z$. At each step, a node j not in S that has the minimum $e_j(S)$ is added to S. Every time a node is added to S, $e_i(S)$ is updated for all nodes i not in S according to the local algorithm of Section 3 (using $e_j(S)$ in place of e_j) with the restriction that only links (i,j) with $j \in S$ are considered. The algorithm terminates when S includes all nodes. The following theorem states that the algorithm correctly solves the minimum-expected-delay problem.

Theorem 3 *The above algorithm computes an optimal routing strategy for Case 2.*

The proof of the theorem will be based on the next two lemmas.

Lemma 2 *There exists a globally optimal strategy ϕ that is locally optimal, i.e., that uses the local strategy ρ_K of Section 3 for each node i, with e_j set to e_j^ϕ for all outgoing links (i,j).*

Proof The proof is similar to that of Theorem 2 in that it is based on the theory of Markov decision processes and the Comparison Lemma. Just as for the local routing problem, the global problem can be formulated as a Markov decision chain. Each state in this formulation corresponds to the particular node at which the packet currently resides and the state of this node's outgoing links. Thus, if there were n nodes in the network and each had exactly m outgoing links, then the total

number of states would be $n2^m$. In each state corresponding to some non-destination node i, the packet can either take one of the "up" links or wait. If it decides to wait, then it will incur a cost of 1 and will randomly transition into one of the states corresponding to node i. The probability of transitioning into a particular one of these states is determined by the probability of the particular configuration of the outgoing links of i. For example, the probability of transitioning into the state with all links being up is $\Pi_j p(i,j)$. Recall that we are assuming link states are independent from slot to slot. If the packet takes link (i,j), then it will incur a cost of $d(i,j)$ and will randomly transition into one of the states corresponding to node j. The probability of transitioning into a particular one of these states is determined by the probability of the particular configuration of outgoing links of j.

For the global problem, a policy is transient if the packet is guaranteed to eventually reach the destination z. The class of transient policies is non-empty since we are assuming that e_i^* is finite for all i. Just as in the proof of Theorem 2, all optimal policies must belong to the class of transient policies, and so we can restrict our attention to transient policies. As in the proof of Theorem 2, it follows that there exists a stationary optimal policy.

Let σ be an optimal global strategy. The stationary optimal policy that uses σ at each step will be denoted σ^∞. Let ϕ denote the global strategy that uses the local strategy ρ_K of Section 3 for each node i, with e_j set to $e_j^\sigma = e_j^*$ for all outgoing links (i,j). Theorem 2 implies that, for any node i, $J(\phi\sigma^\infty)(x) \leq J(\sigma^\infty)(x)$ for all states x for which the packet is at node i. By the Comparison Lemma, this implies that $J(\phi^\infty)(x) \leq J(\sigma^\infty)(x)$ for all x. Since σ is optimal, $J(\phi^\infty)(x) = J(\sigma^\infty)(x)$ for all x. Thus, ϕ is optimal and $e_i^\phi = e_i^*$ for all i. This implies that ϕ is locally optimal, which proves the theorem. ■

The next lemma shows that $e_i^* > e_j^*$ for any link (i,j) that is used by a strategy that is globally and locally optimal (i.e., for any link (i,j) that is one of the $K = K(i)$ best links for node i in such a strategy). This will allow us to efficiently solve the routing problem for Case 2, despite the existence of cycles in the graph G. To better appreciate this result, we give an example showing that this inequality does not hold even for Case 1.

Example Consider the network shown in Figure 2, modified to exclude link $(2,1)$, and let $d(i,j) = 1$ for all links. Let $p(1,z) = .9$ and $\lambda(1,z) = .1$. Thus, the link is up 90% of the time, but when it goes down it takes an average of 10 slots to come back up. If node 1 used only this link, then from Equations (2) and (3) we compute $e_1(1) = 2$ and $f_1(1) = 11$. Suppose $e_2 = 9$. Then Theorem 1 implies that node 1 should also use the link $(1,2)$, since $e(1,2) = 1 + e_2 < 11$. Since adding this link would only decrease e_1, we have $e_1 < e_2$.

The proof of the next lemma uses the following observation. By comparing Equations (2) and (3), we see that $f_i(K) = 1 + e_i(K)$. This makes intuitive sense, since if the K best links are currently down, then after one time slot the expected delay is the same as for steady state (because of the independence assumption).

Lemma 3 *Suppose $p(i,j) = \lambda(i,j)$ and $d(i,j) \geq 1$ for all links (i,j) (Case 2). Let ϕ be the locally and globally optimal strategy described in Lemma 2. Then $e_i^* > e_j^*$ for any link (i,j) that is used by strategy ϕ (i.e., is one of the $K(i)$ best links for node i).*

Proof Let $k \leq K(i)$ and let (i,j_k) be the kth best link for node i. Then by the condition of Theorem 1, $f_i(k-1) > e(i,j_k)$. Therefore, by Lemma 1, $f_i(k-1) > f_i(k)$. We next show that $f_i(k) > e(i,j_k)$. Suppose to the contrary that $f_i(k) \leq e(i,j_k)$. Then, substituting Equations (7) and (8) into Equation (4) gives

$$f_i(k) = \frac{R_{k-1} + L_{k-1}\lambda(i,j_k)e(i,j_k)}{1 - L_{k-1}(1-\lambda(i,j_k))} \quad (17)$$

$$\geq \frac{R_{k-1} + L_{k-1}\lambda(i,j_k)f_i(k)}{1 - L_{k-1}(1-\lambda(i,j_k))} , \quad (18)$$

which implies

$$f_i(k) \geq \frac{R_{k-1}}{1 - L_{k-1}} = f_i(k-1) . \quad (19)$$

Since this contradicts $f_i(k-1) > f_i(k)$, we have proved that $f_i(k) > e(i,j_k)$. In particular, $f_i(K) > e(i,j_K)$. From the discussion above,

$$f_i(K) = 1 + e_i(K) = 1 + e_i^* , \quad (20)$$

and so $1 + e_i^* > e(i,j_K)$. Also, for all $k \leq K$,

$$e(i,j_K) \geq e(i,j_k) = d(i,j_k) + e_{j_k}^* \geq 1 + e_{j_k}^* . \quad (21)$$

Therefore, $e_i^* > e_{j_k}^*$, proving the lemma. ■

Proof of Theorem 3 It suffices to show that $e_i(S) = e_i^*$ for all nodes when the algorithm terminates. To this end, we prove by induction that, each time a node i is added to the set S, $e_i(S) = e_i^*$ and node i minimizes e_k^* among all nodes k outside of S. The first node added to S is the destination z, and $e_z(S) = e_z^* = 0$. Now suppose that the inductive hypothesis holds for the first m nodes added to S. Let L denote the set of nodes outside S with the smallest e_j^*. (L will have one element if there are no ties, and more than one otherwise.) Let j be any node in L. Then by the inductive hypothesis and Lemma 3, there exists an optimal strategy such that node j uses only links (j,k)

with $k \in S$. But $\epsilon_j(S)$ is the minimum expected delay for node j given that it can only use links (j,k) with $k \in S$. Therefore, $\epsilon_j(S) = \epsilon_j^*$, which is thus true for all nodes in L. Let i denote the next node added to S. Then $\epsilon_i(S) \leq \epsilon_j(S)$. But $\epsilon_i^* \leq \epsilon_i(S)$, because ϵ_i^* is the minimum expected delay for node i over all strategies. Therefore, $\epsilon_i^* \leq \epsilon_i(S) \leq \epsilon_j(S) = \epsilon_j^*$. Therefore, since $j \in L$, node i minimizes ϵ_k^* among all nodes k outside of S. Thus, node i is also in L, and so $\epsilon_i(S) = \epsilon_i^*$. By induction, the theorem is proved. ■

We now analyze the time complexity of the algorithm for Case 2. For each node i not in S, the local algorithm of Section 3 must be performed whenever j is added to S for some outgoing link (i,j). Thus, for each node i, the local algorithm must be performed at most Δ_i times, where Δ_i is the number of outgoing links at node i. Recall that the local algorithm requires $O(\Delta_i)$ time, assuming that the outgoing links are already sorted. Since only one new outgoing link (i,j) is added each time the local algorithm is performed for node i, sorting can also be done in $O(\Delta_i)$ time. Thus, the total time required for performing the local algorithm is $O(\sum_i \Delta_i^2)$. In addition, finding the node of minimum $\epsilon_i(S)$ to add to S at each step requires a total time of $O(|E| \log_{(1+|E|/|V|)} |V|)$ using a d-heap [5]. This latter time complexity is the same as that of the implementation of Dijkstra's shortest-path algorithm using a d-heap [5]. Therefore, the total running time of the algorithm for Case 2 is $O(\sum_i \Delta_i^2 + |E| \log_{(1+|E|/|V|)} |V|)$. Defining Δ to be the maximum Δ_i, we have

$$\sum_i \Delta_i^2 \leq \Delta \sum_i \Delta_i = |E|\Delta \quad . \tag{22}$$

Therefore, the running time can also be expressed as $O(|E|[\Delta + \log_{(1+|E|/|V|)} |V|])$.

If we assume that each link delay $d(i,j)$ is 1 (or that all link delays are equal), then we can obtain an improved time complexity. The reason is that under this assumption, $e(i,j)$ (defined in Section 3) is equal to $1 + e_j$, and so the ordering of node i's outgoing links (i,j) depends only on e_j. Since in the above algorithm, nodes are added to S in order of increasing ϵ_j, it follows that the new link (i,j) that node i is allowed to use when node j is added to S is the worst link that node i is allowed to use. Therefore, rather than running the local algorithm completely when node i obtains a new link, we can maintain the variables R_k and L_k for each node i and update them according to (7) and (8) whenever node i obtains a new link. In other words, the local algorithm is executed only once for each node, but this execution may be spread over several steps of the above algorithm. Moreover, since the outgoing links are already sorted, the local algorithm requires only $O(\Delta_i)$ time for each node i (see Section 3). It follows that the total time required to perform the local algorithm for all nodes is $O(\sum_i \Delta_i) = O(|E|)$. As discussed above, an additional total time of $O(|E| \log_{(1+|E|/|V|)} |V|)$ is required for finding the node of minimum $\epsilon_i(S)$ to add to S at each step. Therefore, if we assume unit link delays, the time complexity of the algorithm for Case 2 is $O(|E| \log_{(1+|E|/|V|)} |V|)$, which is the same as for Dijkstra's shortest-path algorithm.

5 Discussion

In this section we discuss heuristics for the general case when G is not a DAG and the link states are not slotwise independent.

One approach is to first select a subgraph of G that is an intuitively good DAG, and then apply the algorithm for Case 1. It is easy to see that there does not exist a DAG that is simultaneously optimal for all source nodes (i.e., that results in minimum expected delay for all nodes when the algorithm for Case 1 is applied). For example, in Figure 2, the DAG containing link $(1,2)$ is usually best for node 1, while the DAG containing link $(2,1)$ is usually best for node 2. The problem of finding a DAG that is optimal for a single source appears difficult.

One way to obtain a DAG is to first run a shortest-path algorithm (using an appropriate link length) to compute the distance $d(i)$ from each node to the destination, and then use the DAG consisting of links (i,j) such that $d(i) > d(j)$. Applying the algorithm for Case 1 to this DAG guarantees a smaller expected delay than shortest-path routing or any other routing scheme restricted to this DAG. In order to guarantee that each node has at least k alternate links to choose from, one can instead compute a k-connected DAG using Topkis' algorithm [6].

Another approach is to simply apply the algorithm for Case 2. Since the link states are not slotwise independent, the routing strategy obtained will not be optimal for the general problem, but it will be optimal for the DAG consisting of links (i,j) such that node i was added to the set S after node j. Although there does not exist a best DAG, this is intuitively the best DAG we have considered.

Another heuristic, which does not involve a DAG, is simply to iterate the dynamic programming equation (2) for all nodes i (using the local algorithm to compute $K(i)$ at each iteration) until ϵ_i converges. This can also be implemented as a distributed algorithm. This algorithm is actually optimal for a model, called Case 3, in which the states of outgoing links of each node are independent for repeated visits of a packet to that node (but otherwise change according to λ and μ). Case 3 lies outside the above model since the Markov assumption is violated on repeated visits of the same node. However, Case 3 includes both Case 1 (in which

repeated visits cannot occur) and Case 2 (in which link states are independent from slot to slot). Simulations show that this heuristic provides a much smaller average delay than a simple alternate-path routing strategy based on shortest paths.

References

[1] D.M. Topkis. A k shortest path algorithm for adaptive routing in communication networks. *IEEE Trans. Communications*, 36(7):855 859, July 1988.

[2] L.G. Valiant. The complexity of enumeration and reliability problems. *SIAM J. Computing*, 8:410–421, 1979.

[3] A.F. Veinott Jr. Markov decision chains. eds. G.B. Dantzig and B.C. Eaves, *MAA Studies in Optimization*, 10:124–159, 1974.

[4] E.L. Lawler. *Combinatorial Optimization: Networks and Matroids*. Holt, Rinehart, and Winston, 1976.

[5] R.E. Tarjan. *Data Structures and Network Algorithms*. SIAM, 1983.

[6] D.M. Topkis. Acyclic k-connected subgraphs for distributed alternate routing in communication networks. *Discrete Applied Mathematics*, 22:181-189, July 1988/89.

ROBUST ROUTING FOR MINIMUM WORST-CASE EXPECTED DELAY IN UNRELIABLE NETWORKS *

Richard G. Ogier and Vlad Rutenburg

SRI International
Menlo Park, CA 94025

Abstract

This paper considers the problem of finding a routing strategy that minimizes the worst-case expected delay from every source to a single destination in an unreliable network, given a constraint on the number of outgoing links at each node that can be inoperational at any point in time. Subject to this constraint, links can fail and recover arbitrarily often. A node having a packet to forward must choose a single link on which to transmit the packet, but does not know in advance which links will be inoperational during the transmission. If a transmission fails, the packet is retransmitted (not necessarily on the same link) after some fixed amount of time. We show that the optimal routing strategy is a stationary randomized strategy in which each node selects the forwarding link according to a fixed probability distribution. We present an efficient algorithm that computes an ϵ-optimal solution in $O(|E|\log(|V|/\epsilon))$ time, for any positive number ϵ.

1 Introduction

This paper considers the important problem of robust routing in unreliable networks whose links can fail and recover frequently and unpredictably. In particular, we consider the problem of finding a routing strategy that minimizes the worst-case expected delay from every source to a single destination in an unreliable network, given a constraint on the number of outgoing links at each node that can be inoperational at any point in time. Subject to this constraint, links are allowed to fail and recover arbitrarily often according to a policy that belongs to a very general class of random, history-dependent policies. The worst case is taken over all policies in this class.

*This work was supported by the Defense Advanced Research Projects Agency and the Rome Air Development Center under Contract F30602-90-C0003, and by the U.S. Army Research Office under Contract DAAL03-88-K-0054.

A node having a packet to forward must select a single link on which to transmit the packet, but does not know in advance which links will be inoperational during the transmission. If the selected link is inoperational, the transmission fails, and the packet is retransmitted (not necessarily on the same link) after a fixed amount of time. We show that the optimal routing strategy is a stationary, history-independent, randomized strategy in which each node selects the forwarding link according to a fixed probability distribution. We present an efficient algorithm that computes an ϵ-optimal solution in $O(|E|\log(|V|/\epsilon))$ time, for any positive number ϵ, where $|V|$ is the number of nodes and $|E|$ is the number of links in the network.

The authors recently presented an algorithm called MEDAR [1] for minimum-expected-delay routing in networks in which links fail and recover according to a two-state Markov chain with known transition probabilities. The present paper is motivated by the desire to achieve robust routing when the link states may not be Markov and may not have known transition probabilities.

In Section 2 we formulate the problem, and in Section 3 we show that the optimal strategies are history independent and stationary. Section 4 presents an algorithm that solves the local problem defined at a single node, and Section 5 presents an algorithm that solves the global problem by applying the local algorithm to each node.

2 Problem Formulation

We assume that the network has an underlying topology given by a directed graph $G = (V, E)$, where V is the set of nodes and E is the set of links. We let z denote the single destination node. Each link in E can switch between two possible states, up (operational) and down (inoperational). Each link (i, j) is assigned a positive number $d(i, j)$ representing the propagation delay of the link. The number $d(i, j)$ can also include

the expected queuing and processing delay at node j, but we assume that $d(i,j)$ is fixed (or quasi-static) and independent of the routing strategy. The problem that allows queuing delay to depend on the routing strategy is important but more complex, so our goal is to first understand the simpler problem, just as it is important to understand shortest-path routing before trying to solve the more general convex-cost network-flow problem.

Since we want to analyze the expected delay of a typical packet, we consider the problem in which there is exactly one packet in the network, initially residing at some nondestination node at time $t = 0$, that must be routed to the destination z. However, our solution provides routing from all nodes to z, and can be used to route an arbitrary number of packets (similarly to shortest-path routing).

At time $t = 0$, and immediately upon arriving at a nondestination node, the node with the packet (say, node i) selects one of its outgoing links (i,j) and transmits the packet on this link. For simplicity, we assume that transmissions occur in zero time. If the selected link is up during the transmission, the packet arrives $d(i,j)$ time units later at node j. If the selected link is down during the transmission, the packet is retransmitted R time units later, possibly on a different link, where R is a fixed positive number representing the time required to recognize that a transmission failed.

Since the link states are important only at packet transmissions, we consider a discrete-time model in which the nth time step ($n = 1, 2, 3, ...$) corresponds to the nth (successful or unsuccessful) transmission of the packet. The position of the packet at the nth time step will be taken to be the transmitting node of the nth transmission. Note that, in each time step in the discrete model, the packet incurs a delay of $d(i,j)$ if the transmission succeeds, and a delay of R if the transmission fails.

We let H_n, $n \geq 1$, denote the history of all packet transmissions and network link states up to and including time step n. We let H_0 specify the node at which the packet initially resides. Note that H_{n-1} determines the position of the packet at time step n. We assume that each node knows H_{n-1} at each time step n, but does not know the current state of its outgoing links or of any link in the network. Thus, a node transmitting the packet at time step n can base its decision on the history up to the preceding time step $n-1$, but not on the current state of its outgoing links. (Although the assumption that node i knows H_{n-1} is impractically strong, we will see that the optimal routing strategy is actually independent of the history.)

We assume that links change state according to the following general model. At each node i, at most $K(i)$ outgoing links can be down at any time, where $K(i)$ is a fixed known nonnegative integer. Since we are considering the worst case, we can assume that *exactly* $K(i)$ outgoing links at node i are down at each time. We let $A(i,n)$ denote the set of $K(i)$ outgoing links at node i that are down at time step n. We assume that $A(i,n)$ is random, and that the probability that $A(i,n) = B$ is $Q_{i,n}(B|H_{n-1})$, where $Q_{i,n}(\cdot|H_{n-1})$ is an unknown probability function that can depend on the history H_{n-1} up to the preceding time step. By the above assumption, $Q_{i,n}(B|H_{n-1})$ can be nonzero only for sets B of size $K(i)$.

For completeness, we assume that the states of the outgoing links at different nodes at time step n are statistically independent, given H_{n-1}. However, this assumption has no effect on the problem, since the packet can be at only one node at any time step.

In terms of control theory, $Q_{i,n} = Q_{i,n}(\cdot|\cdot)$ is a decision rule that randomly selects the states of the outgoing links of node i at time step n, based on the previous network history. Let Q denote the collection of decision rules $Q_{i,n}$ for all i and n. We will call Q the *environment*, since it determines how link states change for all time. For convenience, we will sometimes call Q a *policy* (the environment policy). The set of all environments that satisfy the above assumptions will be denoted $\mathcal{E}$. (We will be considering the worst case over all $Q \in \mathcal{E}$.)

Clearly, the routing strategy that minimizes the worst-case expected delay must be random, since in the worst case for any deterministic routing strategy, the link selected for transmission will always fail. We thus let $P_{i,n}(j|H_{n-1})$ denote the probability that node i will select the outgoing link (i,j) at time step n, given the history H_{n-1}. Thus, $P_{i,n} = P_{i,n}(\cdot|\cdot)$ is a decision rule used by node i to randomly select an outgoing link for time step n, based on the previous history. Let P denote the collection of decision rules $P_{i,n}$ for all i and n. We will call P the *routing policy*. The set of all routing policies that satisfy the above assumptions will be denoted $\mathcal{R}$.

We consider the following *minimax expected-delay routing problem*. Given G, $K(i)$ for all i, $d(i,j)$ for all (i,j), and the above assumptions, find a routing policy that minimizes the worst-case expected delay for a packet to reach the destination z from any source. We let $e(i,P,Q)$ denote the expected delay (in actual time, not time steps) for the packet to reach z, given that it initially resides at node i, when the routing policy is P and the environment is Q. The problem can be stated mathematically as follows:

$$\min_{P \in \mathcal{R}} \max_{Q \in \mathcal{E}} e(i,P,Q) \quad \text{for all } i \neq z. \qquad (1)$$

The value of the above minimum for node i is denoted

$e^*(i)$, and is called the minimax (or minimum worst-case) expected delay from node i to z.

We will actually compute a routing strategy P^* and an environment Q^* such that, for every node $i \neq z$, (P^*, Q^*) is a saddle point of the function $e(i, P, Q)$. That is, for all routing policies $P \in \mathcal{R}$ and environments $Q \in \mathcal{E}$, and for all $i \neq z$,

$$e(i, P^*, Q) \leq e(i, P^*, Q^*) \leq e(i, P, Q^*) \quad . \qquad (2)$$

(We are thus solving a two-person game.) It is clear that if (P^*, Q^*) satisfies (2), then P^* achieves the minimum in (1) and thus solves the minimax expected-delay routing problem.

We need to make an assumption on the compatibility between the topology G and $K(i)$ to ensure that the minimum worst-case expected delay $e^*(i)$ is finite for all nodes i. A directed acyclic graph (DAG) is a directed graph that has no directed cycles. A DAG is said to be destination oriented if the destination is the only node that has no outgoing links in the DAG. We assume that the graph G has a subgraph that is a destination-oriented DAG such that each node $i \neq z$ has at least $K(i)+1$ outgoing links in the DAG. Since at most $K(i)$ outgoing links at node i can be down at any point in time, this assumption ensures that a packet originating at any node can reach z even in the worst case. For example, the routing policy in which node i selects each outgoing DAG link with the same probability results in a finite expected delay for all nodes and for any environment.

Note in particular that $K(i)$ must be equal to zero for at least one node i that is a neighbor of z, since otherwise all links (i, z) coming into the destination can be down at all times, preventing any packet from reaching z.

3 Restriction to Memoriless Policies

In this section, we will show that the optimal routing policy P^* and environment Q^* are memoriless (history independent) and stationary (time independent). This will allow us to restrict our attention to such policies in the remainder of the paper.

A memoriless and stationary routing policy P is such that for each node i and time step n, $P_{i,n}(j|H_{n-1})$ is independent of n and H_{n-1}, and can thus be written $P_i(j)$. We will use the simpler notation $p(i, j)$. Thus $p(i, j)$ is the probability that node i selects the outgoing link (i, j) at any time step. (For convenience, we assume that all nodes select an outgoing link at each time step, so that the policy is independent of the packet location.) We let $\mathcal{R}'$ denote the set of policies in $\mathcal{R}$ that are memoriless and stationary.

Similarly, a memoriless and stationary environment Q is such that $Q_{i,n}(B|H_{n-1})$ is independent of n and H_{n-1}, and can thus be written $Q_i(B)$. We will let $q(i, j)$ denote the probability that link (i, j) is down at any time step. Since exactly $K(i)$ outgoing links at node i are down at any time step, we have $\sum_j q(i, j) = K(i)$. Since node i can select only one outgoing link at any time step, the joint probability that two or more given outgoing links are down has no effect on the problem. Therefore, the probabilities $q(i, j)$ for all j are sufficient for representing the complete probability function $Q_i(\cdot)$. We let $\mathcal{E}'$ denote the set of environments in $\mathcal{E}$ that are memoriless and stationary.

If P and Q are memoriless, then we have a Markov chain, denoted $M(P, Q)$, whose state is the current position of the packet. If P and Q are also stationary, then this Markov chain is stationary with state transition probability $p(i, j)[1 - q(i, j)]$ from node i to node j. The cost incurred for such a transition is $d(i, j)$, and the cost incurred for a transition from node i to itself is R. If Q is fixed, then $M(P, Q)$ is a controlled Markov chain with control policy P. Similarly, if P is fixed, then $M(P, Q)$ is a controlled Markov chain with control policy Q.

We let p and q denote the memoriless stationary routing and environment policies represented by the probabilities $p(i, j)$ and $q(i, j)$. We consider the *restricted problem* of finding memoriless stationary policies p^* and q^* such that, for all $p \in \mathcal{R}'$ and $q \in \mathcal{E}'$, and for all $i \neq z$,

$$e(i, p^*, q) \leq e(i, p^*, q^*) \leq e(i, p, q^*) \quad . \qquad (3)$$

The existence of a solution (p^*, q^*) to the restricted problem will be established in Section 5 (independently of the results of this section). The following theorem states that any solution to the restricted problem is also a solution to the unrestricted problem (2), and therefore to the minimax expected-delay routing problem. The remainder of the paper will therefore consider only the restricted problem.

Theorem 1 *Suppose a pair of policies (p^*, q^*) solves the restricted problem (3). Then it also solves the unrestricted problem (2).*

Proof. We need to show that, for any $P \in \mathcal{R}$ and $Q \in \mathcal{E}$, and for all $i \neq z$,

$$e(i, p^*, Q) \leq e(i, p^*, q^*) \leq e(i, P, q^*) \quad . \qquad (4)$$

We shall prove the second inequality; the proof of the first inequality is similar. To this end, suppose q^* is used. It suffices to show that there exists an optimal

routing policy that is memoriless and stationary (since by (3) p^* is optimal over all such policies). Since q^* is memoriless, the past history H_{t-1} provides no information about the future link states, and so it is clear that the optimal routing policy is memoriless. We therefore have a controlled Markov chain $M(P, q^*)$ with control policy P, as discussed above. Since q^* is stationary, it follows from the theory of controlled Markov chains that there exists an optimal stationary routing policy ([3, page 298] or [2, page 140]). Since this policy is also memoriless, the second inequality of (4) is established. ■

4 Solution to the Local Problem

In this section, we consider a fixed nondestination node i. For each outgoing link (i,j) of node i, we let $e(i,j)$ be a fixed given positive number. (In the next section, $e(i,j)$ will be taken to be $d(i,j) + e(j,p,q)$ for some global policies p and q.) Since i is fixed in this section we will write K in place of $K(i)$.

In the local problem, a routing policy is represented by a probability $p(i,j)$ assigned to each outgoing link (i,j). We assume that node i initially has a packet to forward. At each time step, node i randomly selects one outgoing link (i,j) according to $p(i,\cdot)$, and transmits the packet on this link. This process is repeated at each time step until the transmission is successful. Since we are considering only memoriless and stationary policies, $p(i,\cdot)$ is the same for all time steps. For every time step in which the selected link is down, the transmission fails and a cost of R is incurred (representing the time between transmissions). When the transmission finally succeeds, a cost of $e(i,j)$ is incurred (representing expected delay to the destination).

As discussed in Section 3, since node i can select only one outgoing link at any time step, the joint probability that two or more given outgoing links are down has no effect on the problem. Therefore, the local environment policy will be represented by a probability $q(i,j)$ assigned to each outgoing link (i,j). Thus, at each time step, the probability that link (i,j) is down is $q(i,j)$. Since exactly K links are down at each time step, we have $\sum_j q(i,j) = K$. (To see this, note that $q(i,j)$ is the expected value of a random variable that is 0 if (i,j) is up and 1 if (i,j) is down. Therefore, the sum of the $q(i,j)$ is the expected number of down links.)

By the kth link we will mean the outgoing link (i,j) with the kth smallest value of $e(i,j)$. If (i,j) is the kth link, we will write e_k, p_k, and q_k in place of $e(i,j)$, $p(i,j)$, and $q(i,j)$. We let M denote the number of outgoing links at node i. We assume $M \geq K+1$, since otherwise the worst-case expected delay is infinite. A local routing policy is thus a vector $p = (p_k : k = 1,...,M)$ and a local environment policy is a vector $q = (q_k : k = 1,...,M)$. The vectors p and q are said to be feasible if they satisfy the following constraints:

$$\sum_k p_k = 1 \tag{5}$$

$$\sum_k q_k = K \tag{6}$$

$$0 \leq p_k \leq 1 \text{ for all } k \tag{7}$$

$$0 \leq q_k \leq 1 \text{ for all } k \ . \tag{8}$$

Let $J(p,q)$ denote the total cost (expected delay) given that the local policies p and q are used. The local problem is to find a saddle point of J, that is, to find policies p^* and q^* such that

$$J(p^*,q) \leq J(p^*,q^*) \leq J(p,q^*) \tag{9}$$

for all feasible p and q.

The above model implies that the expected delay $J(p,q)$ using policies p and q satisfies the following recursive equation:

$$J(p,q) = \sum_{k=1}^{M} p_k(1-q_k)e_k + (\sum_{k=1}^{M} p_k q_k)(R + J(p,q)) \ . \tag{10}$$

The kth term of the first sum is the probability that the kth link is selected and is up, times the expected delay using the kth link. The second sum is the probability that the selected link is down. The factor $R + J(p,q)$ represents the fact that if the selected link is down, a retransmission delay of R is incurred, and the expected remaining delay at the next time step is unchanged.

Note that $J(p,q)$ is a solution of the equation $x = f(p,q,x)$, where

$$f(p,q,x) \stackrel{\text{def}}{=} \sum_{k=1}^{M} p_k(1-q_k)e_k + (\sum_{k=1}^{M} p_k q_k)(R + x) \ . \tag{11}$$

Although Equation (10) is easily solved for $J(p,q)$, the proofs of optimality are made simpler by considering $f(p,q,x)$.

A pair (p,q) of feasible policies is said to be *transient* if $\sum_{k=1}^{M} p_k q_k < 1$, or equivalently, if there is a nonzero probability that the selected link is up. Thus, if a transient pair of policies is used, then the packet will eventually be successfully transmitted on some link.

Lemma 1 *Let (p,q) be a transient pair of policies. Then the equation $x = f(p,q,x)$ has a unique solution*

$J(p,q)$. *Furthermore,*

$$x \le f(p,q,x) \;\Rightarrow\; x \le J(p,q) \;, \qquad (12)$$

$$x \ge f(p,q,x) \;\Rightarrow\; x \ge J(p,q) \;. \qquad (13)$$

Proof. Note that the function $x \mapsto f(p,q,x)$ is a line with slope $\sum_k p_k q_k < 1$. Therefore, this line must intersect the line $x \mapsto x$ at exactly one point x, implying that the equation $x = f(p,q,x)$ has a unique solution. Moreover, since the the slope of the first line is less than the slope of the second line, it follows that the first line is above the second line to the left of where they intersect, and is below the second line to the right of where they intersect, which establishes (12) and (13). ■

The following lemma implies that, in order to prove that (p,q) is a saddle point of J, it suffices to show that, for some x, $x = f(p,q,x)$ and (p,q) is a saddle point of $f(\cdot,\cdot,x)$.

Lemma 2 *Let (p,q) be a transient pair of policies, and let $x = J(p,q)$, i.e., $x = f(p,q,x)$. Suppose that*

$$f(p,q',x) \le x \le f(p',q,x) \qquad (14)$$

for all feasible policies p' and q'. Then (p,q) is a saddle point of J.

Proof. By Lemma 1, (14) implies that $J(p,q') \le x \le J(p',q)$, showing that (p,q) is a saddle point of J. ■

The next lemma reduces the problem of finding a saddle point (p^*,q^*) of J to the problem of solving a single equation for the optimal value $x^* = J(p^*,q^*)$. For any $x > 0$, we define $m(x)$ to be the largest integer k such that $e_k < x$. For any $x > e_{K+1}$, we define the pair of vectors p and q *generated by* x as follows: $p_k = q_k = 0$ for $k > m(x)$, and for $k \le m(x)$,

$$p_k = \frac{R}{m(x)-K} \cdot \frac{1}{x+R-e_k} \;, \qquad (15)$$

$$q_k = 1-(m(x)-K)p_k = \frac{x-e_k}{x+R-e_k} \;. \qquad (16)$$

Lemma 3 *Let $x > e_{K+1}$, and let p and q be the vectors generated by x. If*

$$\sum_{k=1}^{m(x)} p_k = 1 \;, \qquad (17)$$

then (p,q) is a saddle point of J, and $x = x^* = J(p,q)$.

Proof. We first show that p and q are feasible. Since $e_{K+1} < x$, we have $m(x) \ge K+1$ by definition. Therefore, $m(x) > K$ and $x > e_k$ for all $k \le m(x)$, and so from (15) and (16) it is clear that p_k and q_k are nonnegative for all k. Since also $\sum_k p_k = 1$, the feasibility of p is established. Now the first equality of (16) implies that $q_k \le 1$ for all k. Finally, summing the first equality of (16) for $k = 1$ to $m(x)$ gives $\sum_k q_k = K$, and so the feasibility of q is established.

From (11), (16), and the fact that $q_k = 0$ for $k > m(x)$, it follows that for any feasible p',

$$f(p',q,x) = \sum_{k=1}^{M} p'_k[e_k + q_k(x+R-e_k)] \qquad (18)$$

$$= \sum_{k=1}^{m(x)} p'_k x + \sum_{k=m(x)+1}^{M} p'_k e_k \qquad (19)$$

$$\ge x \;, \qquad (20)$$

where the last inequality follows from the facts $\sum_{k=1}^{M} p'_k = 1$ and $e_k \ge x$ for $k > m(x)$. If p' is such that $p'_k = 0$ for $k > m(x)$, then the last inequality can be replaced by equality. In particular, $f(p,q,x) = x$, i.e., $x = J(p,q)$.

Similarly, using (15) and routine manipulations, for any feasible q',

$$f(p,q',x) = x - \frac{R}{m(x)-K}[K - \sum_{k=1}^{m(x)} q'_k] \qquad (21)$$

$$\le x \;, \qquad (22)$$

where the inequality uses the fact that $\sum_{k=1}^{m(x)} q'_k \le K$.

We have shown that for all feasible policies p' and q', $f(p,q',x) \le x \le f(p',q,x)$ and that $x = J(p,q)$. Therefore, by Lemma 2, (p,q) is a saddle point of J. ■

The above lemma reduces the problem of finding a saddle point of J to the problem of solving the following equation for $x = x^*$:

$$1 = g(x) \stackrel{\text{def}}{=} \frac{R}{m(x)-K} \sum_{k=1}^{m(x)} \frac{1}{x+R-e_k} \;. \qquad (23)$$

An example of the function $g(\cdot)$ is given in Figure 1, where $M = 5$, $K = 1$, $R = 1$, $e_1 = 2$, $e_2 = 3$, $e_3 = 3.2$, $e_4 = 4$, and $e_5 = 4.5$. The solution x^* to $g(x) = 1$ is approximately 3.37, which lies between e_3 and e_4, implying that $m(x^*) = 3$. Therefore, the optimal routing policy uses 3 of the 5 outgoing links. From (15) we find that p_1, p_2, and p_3 are approximately 0.21, 0.36, and 0.43, respectively, and from (16) we find that q_1, q_2, and q_3 are approximately 0.58, 0.27, and 0.15, respectively.

Note that each term of the sum in (23) is continuous and decreasing in x, but since $m(x)$ jumps at each point $x = e_k$, $g(x)$ may be discontinuous at these points. As defined, $m(x)$ is left continuous. Therefore,

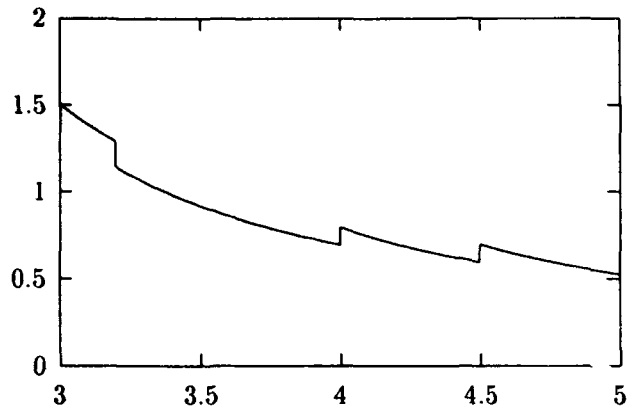


Figure 1: An Example of the Function $g(x)$

$g(x)$ is left continuous, and is continuous and decreasing on each interval $(e_k, e_{k+1}]$ for $k = K+1, ..., M$, where we take e_{M+1} to be infinity. We define $g(x+)$ to be the limit of $g(y)$ as y decreases to x. From (23) it is easily verified that $g(e_{K+1}+) > 1$ and that $g(x)$ approaches 0 as x goes to infinity. However, since $g(\cdot)$ is not continuous, these facts do not imply that $g(x^*) = 1$ for some x^*. The following lemma establishes the existence of x^*, and gives properties of $g(\cdot)$ that will allow us to approximate x^* using binary search.

Lemma 4 *There exists a unique $x^* > e_{K+1}$ such that $g(x^*) = 1$. Moreover, $g(x) > 1$ for all $x < x^*$, and $g(x) < 1$ for all $x > x^*$.*

Proof. We know that $g(\cdot)$ is continuous and decreasing on each interval $(e_k, e_{k+1}]$, so we will compute the difference $g(x+) - g(x)$ at each point $x > e_{K+1}$ such that $x = e_k$ for some k. To this end, suppose that $e_{k+1} > e_k$ for some $k \geq K+1$, and let v be the number of links that have the delay e_{k+1}, i.e., such that $e_{k+1} = e_{k+v} < e_{k+v+1}$. Then, letting $x = e_{k+1}$, we have $m(x) = k$ and $m(x+) = k+v$. Substituting these values into (23) and using routine manipulations, we obtain

$$g(x+) - g(x) = \frac{v(1 - g(x))}{k + v - K} \quad . \tag{24}$$

Since $0 < v/(k+v-K) < 1$, this equation implies that if $g(x) > 1$, then $g(x) > g(x+) > 1$, and if $g(x) < 1$, then $g(x) < g(x+) < 1$. (Figure 1 illustrates these facts.) Thus, $g(x)$ never crosses 1 at a discontinuity. Therefore, since $g(e_{K+1}+) > 1$, $g(x) \to 0$ as $x \to \infty$, and $g(\cdot)$ is decreasing where it is continuous, the lemma follows. ■

Suppose we make a guess x as to what x^* is. Then by the above lemma, if $g(x) > 1$, our guess was too small, and if $g(x) < 1$, our guess was too large. Therefore, if we know an upper bound for x^*, then we can approximate x^* using binary search. The following lemma provides this upper bound.

Lemma 5 *The optimal value x^* satisfies $e_{K+1} < x^* \leq e_{K+1} + RK$.*

Proof. The first inequality has already been established, so we need to show that $x^* \leq e_{K+1} + RK$. Consider the routing policy $p_k = 1/(K+1)$ for $k = 1, ..., K+1$. Since at most K links are down at any time, this policy has a probability of success of at least $1/(K+1)$. Therefore, the packet can achieve a successful transmission on one of the best $K+1$ links after an average of $K+1$ tries, implying $x^* \leq e_{K+1} + RK$. ■

We next present an algorithm for finding x such that $|x - x^*| < \delta$ for any given positive number δ. Although we can perform binary search on the interval $(e_{K+1}, e_{K+1} + RK]$ to find such an x, it will be convenient to first find $m(x^*)$, which we will denote m^*. By the properties of $g(\cdot)$ that have been established, it is clear that m^* is the largest k such that $g(e_k+) > 1$. Since evaluating $g(e_k+)$ requires $O(M)$ time, m^* can be found in $O(M^2)$ time. However, using binary search, the time complexity can be reduced to $O(M \log M)$.

After computing the integer m^*, we know that x^* lies in the interval $(e_{m^*}, b]$, where $b = \min\{e_{m^*+1}, e_{K+1} + RK\}$. Since the length of this interval is no greater than RK, we can use binary search to find an x such that $|x - x^*| < \delta$ in $\log(\frac{RK}{\delta})$ iterations. Since R is a constant (i.e., $R = O(1)$), the search requires $O(\log(K/\delta))$ iterations. Since each iteration requires $O(M)$ time (to compute the right side of Equation (23)), the algorithm requires $O(M \log(K/\delta))$ time. Since the algorithm for finding m^* required $O(M \log M)$ time, and since $K < M$, the total running time is $O(M \log(M/\delta))$.

Let x^+ denote the approximation of x^* found by binary search. We assume that $x^+ \leq x^*$ (i.e., we let x^+ be the last number found by the search that was less than or equal to x^*). Since the p_k given by (15) with $x = x^+$ do not sum to exactly 1, we do not yet have a solution. We therefore modify p, q to p^+, q^+ as follows: $p_k^+ = q_k^+ = 0$ for $k > m^*$, and for $k \leq m^*$,

$$p_k^+ = \frac{R - \alpha}{m^* - K} \cdot \frac{1}{x^+ + R - e_k} \quad , \tag{25}$$

$$q_k^+ = 1 - (m^* - K)p_k^+ = \frac{x^+ + \alpha - e_k}{x^+ + R - e_k} \quad , \tag{26}$$

where α is a positive (easily computed) number chosen such that $\sum_k p_k^+ = 1$. Using the facts that $|x^+ - x^*| < \delta$ and that (23) holds for $x = x^*$, it is is easily verified that $\alpha \leq \delta$.

We will show that the pair (p^+, q^+) is an approximate saddle point of J, but we first need to define what this means. A pair (p', q') of feasible vectors is said to be an ϵ-approximate saddle point of J if

$$J(p', q) - \epsilon \le J(p', q') \le J(p, q') + \epsilon \quad , \qquad (27)$$

for all feasible p and q. If (p', q') is an ϵ-approximate saddle point of J, we say that p' and q' are ϵ-optimal policies for the local problem. The following lemma implies that if (p', q') is an ϵ-approximate saddle point of J, then $J(p', q)$ is guaranteed to be no greater than $J(p^*, q^*) + 2\epsilon$, and $J(p, q')$ is guaranteed to be no less than $J(p^*, q^*) - 2\epsilon$, where (p^*, q^*) is a saddle point of J.

Lemma 6 *Suppose (p', q') satisfies (27). Then*

$$J(p', q) - 2\epsilon \le J(p^*, q^*) \le J(p, q') + 2\epsilon \quad , \qquad (28)$$

for all feasible p and q.

Proof. It suffices to show that $|J(p', q') - J(p^*, q^*)| \le \epsilon$. The second inequality in (27) with $p = p^*$ implies that $J(p', q') \le J(p^*, q') + \epsilon$. But since (p^*, q^*) is a saddle point of J, $J(p^*, q') \le J(p^*, q^*)$. Therefore $J(p', q') \le J(p^*, q^*) + \epsilon$. A similar argument shows that $J(p', q') \ge J(p^*, q^*) - \epsilon$, and so $|J(p', q') - J(p^*, q^*)| \le \epsilon$. ■

The following lemma allows one to prove approximate relations between x and $J(p, q)$ by proving approximate relations between x and $f(p, q, x)$.

Lemma 7 *Let (p, q) be a transient pair of policies, and let a be a number such that $0 < a < 1$ and $\sum_k p_k q_k \le a$. Then for any $\epsilon > 0$,*

$$x \le f(p, q, x) + (1 - a)\epsilon \quad \Rightarrow \quad x \le J(p, q) + \epsilon \quad ,$$
$$x \ge f(p, q, x) - (1 - a)\epsilon \quad \Rightarrow \quad x \ge J(p, q) - \epsilon \quad .$$

Proof. Let $y = J(p, q)$, i.e., $y = f(p, q, y)$. Recall that the function $x \mapsto f(p, q, x)$ is a line with slope $\sum_k p_k q_k \le a$. Thus, the function $x \mapsto x - f(p, q, x)$ is a line with slope no less than $1 - a$. Therefore, since $J(p, q) - f(p, q, J(p, q)) = 0$, if $x > J(p, q) + \epsilon$, then $x - f(p, q, x) > (1 - a)\epsilon$. Equivalently, if $x - f(p, q, x) \le (1 - a)\epsilon$, then $x \le J(p, q) + \epsilon$, proving the first statement of the lemma. The second statement is proved similarly. ■

We can now present the main result of this section.

Theorem 2 *The pair (p^+, q^+) defined by (25) and (26) is an ϵ-approximate saddle point of J with $\epsilon = (K + 1)\alpha$. It follows that, for any $\epsilon > 0$, an ϵ-approximate saddle point can be computed in $O(M \log(M/\epsilon))$ time.*

Proof. We first compute $\sum_k p_k^+ q_k^+$, so that we can apply Lemma 7. Using the first equality in (26) and the fact that p_k^+ and q_k^+ are zero for $k > m^*$, we have

$$\sum_{k=1}^{M} p_k^+ q_k^+ = \sum_{k=1}^{m^*} p_k^+ [1 - (m^* - K) p_k^+] \qquad (29)$$

$$= 1 - (m^* - K) \sum_{k=1}^{m^*} (p_k^+)^2 \qquad (30)$$

$$\le 1 - \frac{m^* - K}{m^*} \sum_{k=1}^{m^*} p_k^+ \qquad (31)$$

$$= \frac{K}{m^*} \le \frac{K}{K + 1} \quad , \qquad (32)$$

where (31) follows from Jensen's inequality.

From (11), (26), and the fact that $q_k^+ = 0$ for $k > m^*$, it follows that for any feasible p,

$$f(p, q^+, x^+) = \sum_{k=1}^{M} p_k [e_k + q_k^+ (x^+ + R - e_k)]$$
$$= \sum_{k=1}^{m^*} p_k (x^+ + \alpha) + \sum_{k=m^*+1}^{M} p_k e_k$$
$$\ge x^+ \quad , \qquad (33)$$

where the last inequality follows from the facts $\sum_{k=1}^{M} p_k = 1$ and $e_k \ge x^+$ for $k > m^*$. Therefore, by Lemma 1, $J(p, q^+) \ge x^+$ for all feasible p.

If p is such that $p_k = 0$ for $k > m^*$, then the inequality (33) can be replaced by $f(p, q^+, x^+) = x^+ + \alpha$. In particular, $f(p^+, q^+, x^+) = x^+ + \alpha$. Therefore, by (32) and the second statement of Lemma 7 with $a = K/(K + 1)$ and $\epsilon = \alpha/(1 - a)$, we have

$$J(p^+, q^+) \le x^+ + \alpha/(1 - a) = x^+ + (K + 1)\alpha \quad . \qquad (34)$$

Therefore, since $J(p, q^+) \ge x^+$ for all feasible p, we have

$$J(p^+, q^+) \le J(p, q^+) + (K + 1)\alpha \qquad (35)$$

for all feasible p. It can be similarly shown that

$$J(p^+, q^+) \ge J(p^+, q) - (K + 1)\alpha \qquad (36)$$

for all feasible q, thus establishing the first statement of the theorem.

Recall that the binary search algorithm used for finding x^+ such that $|x^+ - x^*| \le \delta$ requires $O(M \log(M/\delta))$ time. Also recall that $\alpha \le \delta$. Thus, if we are given ϵ, then by choosing $\delta = \epsilon/(K + 1)$, we obtain an ϵ-approximate saddle point (p^+, q^+) in $O(M \log(M/\epsilon))$ time, where we used the fact that $O(\log MK) = O(\log M + \log K) = O(\log M)$. ■

5 Global Solution

In this section, we show that an ϵ-optimal global policy can be obtained by applying the local algorithm once to each node, in order of increasing expected delay to the destination. The global algorithm resembles Dijkstra's algorithm for finding shortest paths.

Global Algorithm

A set S of nodes is maintained; initially S is empty. Each node i maintains a variable $e_i(S)$, which is an estimate of the optimal expected delay $e^*(i)$. Initially, $e_z(S) = 0$ and $e_i(S) = \infty$ for $i \neq z$. At each step, a node j not in S that has the minimum $e_j(S)$ is added to S. Every time a node is added to S, the local algorithm is executed for each i not in S (in any order), with $e(i,j) = d(i,j) + e_j(S)$ for all outgoing links (i,j) with $j \in S$, and with all outgoing links (i,j) with $j \notin S$ removed from consideration. Upon completion of the the local algorithm for node i, the probabilities $p(i,j)$ and $q(i,j)$ for all outgoing links (i,j) are set equal to those computed by the local algorithm, and $e_i(S)$ is set equal to the value $J(p,q)$ computed by the local algorithm. The algorithm terminates when S includes all nodes.

Complexity. Recall that the local algorithm requires $O(M \log(M/\epsilon))$ time, where M is the number of outgoing links. The above algorithm applies the local algorithm once to each node, thus requiring $O(|E| \log(M/\epsilon))$ time. In addition, finding the node of minimum $e_i(S)$ to add to S at each step requires a total time of $O(|E| \log |V|)$ using a d-heap [4]. Therefore, the total running time of the algorithm is $O(|E|(\log(M/\epsilon) + \log|V|))$. Since $M < |V|$, the time complexity is $O(|E| \log(|V|/\epsilon))$

To simplify the presentation, we first assume that the local algorithm computes optimal local strategies $p(i,\cdot)$ and $q(i,\cdot)$, and show that the global algorithm computes optimal global strategies p and q. We later show that if $p(i,\cdot)$ and $q(i,\cdot)$ are ϵ-optimal for each i, then p and q are $|V|\epsilon$-optimal global strategies.

Theorem 3 *Let p^* and q^* be the policies computed by the global algorithm, assuming that the local algorithm computes optimal policies. Then p^* and q^* are optimal for the restricted problem (3) and therefore for the minimax expected-delay routing problem. When the global algorithm terminates, $e_i(S) = e(i,p^*,q^*)$ for all i.*

Our proof of optimality closely follows that of the MEDAR algorithm [1] for the case in which link states are independent for different time steps. The main difference is that in this paper we need to show that two (not one) policies are simultaneously optimal. That is, to to show that policies p^* and q^* are optimal, we need to show that for each node i, p^* minimizes $e(i,p,q^*)$ over all routing policies p, and that q^* maximizes $e(i,p^*,q)$ over all environment policies q.

The problem of minimizing $e(i,p,q^*)$ over all routing policies p is a stochastic control problem with an infinite horizon and an undiscounted cost function (e.g., [3]). The following lemma uses well-known necessary and sufficient optimality conditions for such problems, to show that there exists a policy that is both locally and globally optimal.

Lemma 8 *There exists a policy p^+ that minimizes $e(i,p,q^*)$ for each i over all policies p, and that uses the optimal local policy of Section 4 for each node i, with $e(i,j)$ set to $d(i,j) + e(i,p^+,q^*)$ for all outgoing links (i,j).*

Proof. The proof is similar to the proof of Lemma 2 of [1], and so we only provide a sketch. Let p' be a global policy that minimizes $e(i,p,q^*)$ for each i. The existence of such a policy follows from stochastic control theory (e.g., [3, page 298] or [2, page 140]). Now let p^+ be the global policy that uses the optimal local policy of Section 4 for each node i, with $e(i,j)$ set to $d(i,j) + e(j,p',q^*)$ for all outgoing links (i,j). Now, for any initial position i of the packet, since the local policy at node i is optimal, if we apply policy p^+ until the packet leaves node i, and apply p' thereafter, the resulting expected delay is the optimal value $e(i,p',q^*)$. It follows from well-known optimality conditions (e.g., [3, page 229] or [2, page 137]) that p^+ is also globally optimal, i.e., $e(i,p^+,q^*) = e(i,p',q^*)$. ■

The following lemma implies that a routing policy p that is locally and globally optimal contains no loops.

Lemma 9 *Suppose p is a locally and globally optimal policy in the sense of Lemma 8. Then $e(i,p,q^*) > e(j,p,q^*)$ for all links (i,j) such that $p(i,j) > 0$.*

Proof. In the notation of Section 4, if we are considering the local problem at node i, then $p_k = p(i,j)$ and $e_k = e(i,j) = d(i,j) + e(j,p,q^*)$, where (i,j) is the outgoing link of node i with the kth smallest value of $e(i,j)$. Also, since $p(i,\cdot)$ is the optimal local policy at node i, the optimal expected delay for the local problem is $x^* = e(i,p,q^*)$. Recall from Section 4 that $p_k > 0$ only if $k \leq m(x^*)$, i.e., only if $x^* > e_k$. That is, $p(i,j) > 0$ only if $e(i,p,q^*) > d(i,j) + e(j,p,q^*) > e(j,p,q^*)$, proving the lemma. ■

Proof of Theorem 3. We need to show that for all policies $p \in \mathcal{R}'$ and $q \in \mathcal{E}'$, and for all $i \neq z$,

$$e(i,p^*,q) \leq e(i,p^*,q^*) \leq e(i,p,q^*) \quad . \qquad (37)$$

We first prove the second inequality. Let e_i^* denote the minimum value of $e(i,p,q^*)$ for all routing policies p. It suffices to show that $e_i(S) = e_i^*$ for all nodes i when the algorithm terminates. To this end, we prove by induction that, each time a node i is added to the set S, $e_i(S) = e_i^*$ and node i minimizes e_k^* among all nodes k outside of S. The first node added to S is the destination z, and $e_z(S) = e_z^* = 0$. Now suppose that the inductive hypothesis holds for the first m nodes added to S. Let L denote the set of nodes outside S with the smallest e_j^*. (L will have one element if there are no ties, and more than one otherwise.) Let j be any node in L. Then by the inductive hypothesis and Lemmas 8 and 9, there exists an optimal strategy such that node j uses only links (j,k) with $k \in S$. But $e_j(S)$ is the minimum expected delay for node j given that it can only use links (j,k) with $k \in S$. Therefore, $e_j(S) = e_j^*$, which is thus true for all nodes in L. Let i denote the next node added to S. Then $e_i(S) \leq e_j(S)$. But $e_i^* \leq e_i(S)$, because e_i^* is the minimum expected delay for node i over all strategies. Therefore, $e_i^* \leq e_i(S) \leq e_j(S) = e_j^*$. Therefore, since $j \in L$, node i minimizes e_k^* among all nodes k outside of S. Thus, node i is also in L, and so $e_i(S) = e_i^*$. By induction, $e_i(S) = e^*(i)$ for all nodes i when the algorithm terminates

We now prove the first inequality of (37). Thus, we assume that the routing policy p^* is used, and we wish to show that the environment policy q^* maximizes $e(i,p^*,q)$ over all environment policies q. Clearly, any optimal environment policy q must satisfy $q(i,j) = 0$ for all links (i,j) such that $p^*(i,j) = 0$. Therefore, we can restrict our attention to the subgraph of G' of G containing only the links (i,j) such that $p^*(i,j) > 0$. Note that $p^*(i,j) > 0$ only for links (i,j) such that the global algorithm added node j to the set S before node i. Therefore, G' is a DAG, i.e., has no cycles. Therefore, letting node i be the kth node to be added to S, the fact that q^* maximizes $e(i,p^*,q)$ follows from the local optimality of q^* and the fact that q^* maximizes $e(j,p^*,q)$ for the first $k-1$ nodes j added to S. Therefore, by induction, q^* maximizes $e(i,p^*,q)$ for all nodes i. ■

The above proof of optimality assumes that the local algorithm computes optimal policies. If the local algorithm computes ϵ-optimal policies, then an error of at most ϵ is incurred at each stage of the global algorithm, and the above proof is easily modified to show that the policies computed by the global algorithm are $|V|\epsilon$-optimal, as indicated by the following proposition.

Proposition 1 *Let p^+ and q^+ be the policies computed by the global algorithm, assuming that the local algorithm computes ϵ-optimal policies. Then p^+ and q^+ are $|V|\epsilon$-optimal, i.e., for all policies $p \in \mathcal{P}'$ and $q \in \mathcal{E}'$, and for all $i \neq z$,*

$$e(i,p^+,q) - |V|\epsilon \leq e(i,p^+,q^+) \leq e(i,p,q^+) + |V|\epsilon \quad . \tag{38}$$

Therefore, if ϵ is replaced by $\epsilon/|V|$, so that the local algorithm computes $\epsilon/|V|$-optimal policies, then the global algorithm computes ϵ-optimal policies in $O(|E|\log(|V|^2/\epsilon)) = O(|E|\log(|V|/\epsilon))$ time.

6 Discussion

We have presented an efficient centralized algorithm for computing a routing strategy that is robust in that it guarantees an upper bound $e(i,p^*,q^*)$ on the expected delay from any node i to the destination in an unreliable network. The network model is such that links are allowed to fail and recover arbitrarily often according to a policy that belongs to a very general class of random, history-dependent policies. We have assumed that the link delays (including any queuing delays) are independent of the routing strategy. The problem that allows queuing delay to depend on the routing strategy is important but more complex, so our goal was to first understand the simpler problem. We remark that a distributed implementation of the algorithm is easily obtained by having each node i apply the local algorithm of Section 4, using estimates of $e(j,p^*,q^*)$ received from neighbors, similar to the Bellman-Ford routing algorithm.

References

[1] R.G. Ogier and V. Rutenburg. Minimum-expected-delay alternate routing. In *Proc. IEEE INFOCOM*, pages 617–625, 1992.

[2] A.F. Veinott Jr. Markov decision chains. *MAA Studies in Optimization*, 10:124–159, 1974. edited by G.B. Dantzig and B.C. Eaves.

[3] D.P. Bertsekas. *Dynamic Programming and Stochastic Control.* Academic Press, 1976.

[4] R.E. Tarjan. *Data Structures and Network Algorithms.* Society for Industrial and Applied Mathematics, 1983.

HOW TO EXTRACT MAXIMUM INFORMATION FROM EVENT-DRIVEN TOPOLOGY UPDATES *

Vlad Rutenburg and Richard G. Ogier

SRI International
Menlo Park, CA 94025

Abstract

This paper addresses the issue of designing the best method for link-state updating in communication networks undergoing topological changes. We conclude that the best way of disseminating link-state information is through event-driven (ED) updating and show how to overcome the traditional drawbacks ED updating. We present an efficient algorithm that extracts the maximum implicit information from ED topology updates by computing the latest time for which each processing node can be certain of the state of each network link. It is shown that the information obtained with this algorithm is equivalent to the information obtained through continuous flooding of link-state information. This paper also presents a method that allows each node to continuously refresh the above information without having to run the above algorithm more than once for each received update. We also present an approximate version of the above algorithm that reduces communication overhead by limiting the radii of propagation.

1 Introduction

One of the most important areas of current research is that of designing packet routing and flow control algorithms in networks that undergo dynamic changes due to the failure and recovery of links. A major determining factor for the success of such algorithms is the timeliness of link state updates, which determines the amount of useful information available to each node (or the operation center) regarding the current and future state of various links in the network. For example, dynamic routing algorithms that base each hop-by-hop routing decision on their probabilistic estimates of the states of the network links and nodes (see, e.g.,[3],[7]), depend on the knowledge of the state of all downstream links in the network. These probabilistic estimates depend on the knowledge of the state of each link at some time t in the past and the conditional distribution of the current link state, conditioned on its state at time t. The more recent this time t is, the better performance of the routing algorithms.

In a network (a directed graph) G, link-state updates (specifying which links are currently operable) are usually distributed by flooding (with intermediate storage) over some spanning subgraph H of G. Throughout this paper we assume that the update packets are assigned the highest routing priority, and that update packets are lost only due to link failures. The two standard approaches to the timing of updates are *periodic updating* (PU), which is self-explanatory, and *event-driven updating* (ED), which involves generating updates in response to changes in link state. The use of PU with long inter-update periods leads to the degradation in the "freshness" of update information. One possible way to assure the best possible "freshness" of updates (within the limitations imposed by the dynamic state of the network) is by using *continuous flooding* (CF) of link-state information (i.e., periodic updating with an infinitesimally short period). However, this approach is extremely wasteful and cannot be implemented in practice.

The above considerations point out the desirability of the ED approach, which is clearly much more efficient and, in fact, very practical. However, we have to answer the question as to how much will the use of ED updating degrade the freshness of link state information, as compared with the idealized CF paradigm. If only a single link is undergoing dynamic changes in the network, it is not difficult to see that the freshness of information about this link under the ED approach will be exactly the same as under the CF approach. However, in general, when all of the links are simultaneously experiencing dynamic changes, the straightforward ED approach provides less information than CF.

*This work was supported by the Defense Advanced Research Projects Agency and the Rome Air Development Center under Contract F30602-90-C0003, and by the U.S. Army Research Office under Contract DAAL03-88-K-0054.

This is due to the fact that, under the ED approach, if a node has not received information about some link for some time, it does not know how recently the state of that link could have changed, since the absence of any new updates could either due to the fact that the state of that link has not changed or to the fact that a new update was blocked by a disconnected network or delayed by a long path.

This paper presents an efficient algorithm that extracts *implicit* information from event-driven updates and computes the latest time for which each node can be certain of the state of each link. Intuitively, this algorithm uses the available information about the state of the links in the network to deduce the real cause for the absence of updates from network links. More precisely, we present algorithms that allow node i to compute the last time T_{jk} for which it is certain of the state of link (j,k). This time, which is usually much more recent than the time contained in the latest update received from link (j,k), can be used by node i to estimate the current and future states of distant links.

Assuming that the update packets (which are of highest routing priority) experience *deterministic* delays over operable links, we will show that the information obtained with our algorithm is *maximum*, in the sense that it is equivalent to the information obtained through the CF approach. Moreover, the same results are extended to the case of *bounded* delays, with the distinction that the information obtained with the algorithm is equivalent to the information obtained through the *worst-case delay* execution of the CF approach.

Numerous papers have been written on the efficient dissemination of link-state information and on reliable broadcast in unreliable networks (e.g.,[1], [2]). However, to the best of our knowledge, no work has been done on the subject of extracting implicit information from updates.

The paper is organized as follows. In Section 2 we introduce our basic model. In Section 3 we present the extrapolation algorithm which allows each node i to compute T_{jk} for each link (j,k) in the network, assuming link-state updates are disseminated using event-driven flooding. Note that T_{jk} can change continuously as a function of time, even between successive topology updates received by node i. Section 4 presents a method that allows each node to continuously refresh T_{jk} without having to run the extrapolation algorithm more than once for each received update. This method is based on the result showing that T_{jk} changes its slope (as a function of time) at most once between two successive topology updates. Section 5 presents extensions to the basic results. Namely, we show how the use of the extrapolation algorithm can further reduce the communication overhead, and how to deal with variable, but bounded link delays. Finally, Section 6 presents an algorithm that computes T_{jk} approximately while reducing the radius of propagation of link-state updates.

2 Basic Model

Let us introduce our basic model. We assume that the clocks are synchronized throughout the network. We assume that each link (i,j) will be subject to breaks and recoveries. When a link breaks (goes down), it becomes inoperable. Moreover, all packets that were in the process of transmission at the time of this event will be lost. Let $d_{i,j}$ denote the total physical delay of link (i,j), when (i,j) is up. For the sake of simplicity, in the original discussion we assume that the delay that each control message will experience at link (i,j), if (i,j) is up, is exactly $d_{i,j}$. Later in the paper, we show how to extend our results to the case of variable but bounded link delays. We also assume that while link (i,j) is down, all the update messages stored at node i, that were supposed to traverse (i,j) and have not lost their relevancy (see Section 5.1) will be instantaneously transmitted as soon as link (i,j) changes its state to "up." Notice that links can be unidirectional or bidirectional and can have different delays in opposite directions, i.e., $d_{i,j} \neq d_{j,i}$. We also assume that the two nodes i and j adjacent to link (i,j) can immediately detect the change of state of (i,j). The case of bounded detection delays can be solved similarly.

We assume that topology updates are flooding along some connected spanning subgraph H of G with link set $E(H)$. The graph H is either equal to the graph G itself or is a proper subgraph of it. The flooding is done as follows. If a link goes down at time t, both of its endpoints send a message, reporting this event and the timestamp t, along each up outgoing link of $E(H)$ immediately, and along each down outgoing link of $E(H)$ as soon as it comes up. However, if a link (i,j) goes up at time t, both of its endpoints send the update message after a waiting period equal to the link delay $d_{i,j}$. The timestamp, nevertheless, should be equal to t. The reason for this waiting is simple: if the link (i,j) remains up for less than the time $d_{i,j}$, which is the time needed to successfully transmit a packet, then this link will not have enough time to deliver any packets, and this situation is the same as if that link did not go up at all.

Let $t^k_{i,j}$ denote the timestamp of the latest update about a change in the state of link (i,j) that has been received by node k, and let $s^k_{i,j}$ denote the state of link (i,j) immediately after that change. The value of $s^k_{i,j}$ is either 1, meaning "up," or 0, meaning "down."

If node k receives an update message about some link (i,j), it compares the timestamp x of that message with the recorded value of $t^k_{i,j}$. If the former is more recent, node k assigns the value of x to the variable $t^k_{i,j}$, and forwards this message along each available (up) outgoing link of $E(H)$ immediately, and along each down outgoing link of $E(H)$ as soon as it comes up (if it is still worth forwarding that message at that time, as discussed in Section 5.1. If the former is more recent, the message is discarded.

3 Extending Link State Information

Let us start with a simple example. Consider a chain a, b, c (see Figure 1). Suppose that node a wants to estimate the state of link (b,c). It knows that its link (b,a) has been down for 2 mseconds. The latest that it heard from link (b,c) was that 15 mseconds ago it went from down to up. However, a knows that since link (b,a) was up 2 mseconds ago, any update from node b that could have reached a at or before 2 mseconds ago, would have done so. Suppose the value of $d_{b,a}$ is 3 mseconds. Then we can deduce that any message that originated from node b at or before time $t^a_{b,a}+d_{b,a} = 2+3 = 5$ mseconds ago, would have reached node a. Thus, node a can deduce that between the time of 15 mseconds ago and the time of 5 mseconds ago link (b,c) was up.

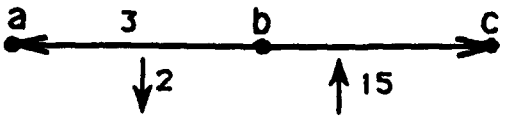


Figure 1: An Example Illustrating the Implicit Information Contained in Link State Updates

In general, let us consider an arbitrary network G. Let T denote the current time. Let us choose an arbitrary node a, and look at the network from a's point of view. We want to compute $T^a_{i,j}$, the latest time in the past such that a can be certain that it has received all the updates about the state of link (i,j) that were generated at or before that time. As long as node a remains fixed, we shall simply write $T_{i,j}$ instead of $T^a_{i,j}$. Let T_i denote the (yet unknown) latest such time that a can be sure that any report generated by node i at or at or before time T_i would have reached node a by the current time T. Let $T^+_{i,j}$ be the latest time that a can be sure that link (i,j) was up. Then clearly

$$\begin{aligned} T^+_{i,j} &= T_{i,j} \quad \text{if} \quad s_{i,j} = 1\,, \\ T^+_{i,j} &= t_{i,j} \quad \text{if} \quad s_{i,j} = 0\,. \end{aligned} \tag{1}$$

We are going to show that the correct values of T_i can be computed by solving the following Bellman equations. Here $N_H(i)$ denotes the set of out-neighbors of i in the update subgraph H, i.e., $N_H(i) = \{j : (i,j) \in E(H)\}$.

$$T_i = \max_{j \in N_H(i)} \{T^+_{i,j} - d_{i,j}\} \text{ for all nodes } i\,, \tag{2}$$

$$T_{i,j} = \max\{T_i, T_j\} \text{ for all links } (i,j)\,. \tag{3}$$

Combining these equations with equation (1) in order to eliminate the variables $T^+_{i,j}$ and $T_{i,j}$, we obtain the following dynamic programming (DP) equation:

$$\begin{aligned} T_i = \max\{ \quad & \max_{\{j \in N_H(i): s_{i,j}=1\}} \{T_j - d_{i,j}\}, \\ & \max_{\{j \in N_H(i): s_{i,j}=0\}} \{t_{i,j} - d_{i,j}\}\} \end{aligned} \tag{4}$$

for all nodes i. Since a node hears from itself immediately, we should also add the boundary condition that

$$T_a = T\,. \tag{5}$$

Intuitively, equation (3) simply says that the state of a link is reported by its endpoints. Equation (2), and thus equation (4), describes the length of the "freshest" path to destination a from node i, i.e., the path among all paths between i and a that could have successfully delivered a packet from i at the most recent time. Equation (4) relates the value of this path length from i to the values of such a length from i's neighbors. We shall prove later (in Theorem 1) that these equations do indeed determine the sought-after values of T_i and $T_{i,j}$. But let us first view the equations (2), (1), (3), (4) and (5) as an abstract set of equations and prove the existence and the uniqueness of their solutions.

By substituting $D_i = T - T_i$ we get:

$$D_a = 0\,. \tag{6}$$

$$\begin{aligned} D_i = \min\{ \quad & \min_{\{j \in N_H(i): s_{i,j}=1\}} \{D_j + d_{i,j}\}, \\ & \min_{\{j \in N_H(i): s_{i,j}=0\}} \{T - t_{i,j} + d_{i,j}\}\} \end{aligned} \tag{7}$$

for all nodes i. First, we are going to prove that the set of equations (7) and (6) have a well-defined and unique solution, by exhibiting a graph N with the property that the equations (7) and (6) are the same as the Bellman equations for the shortest-path problem in N. The node set for N will be the same as for G. The construction is difficult because the value of D_i in (7) maximizes not only over the values of other "shortest distance" variables D_j's but over a mixture of variables D_j and constants $t_{i,j}$. This is all taken care of by defining the following edge set of N with corresponding link "lengths": For every edge $(i,j) \in E(H)$ with $s_{i,j} = 1$,

create a directed link (v_i, v_j) of length $d_{i,j}$. For every edge $(i,j) \in E(H)$ with $s_{i,j} = 0$, create a directed link (v_i, v_a) of length $T - t_{i,j} + d_{i,j}$. It is now a routine exercise to verify that if we let D_i denote the shortest distance from v_i to v_a, then the Bellman equation (see, for example [4]) for these variables D_i is given by (7) and (6). Thus, we have proven the following lemma.

Lemma: The shortest-path problem with the node v_a as the destination in network N, defined above, has its Bellman equations given by (7) and (6).

Having succeeded in visualizing the equations (7) and (6) as describing the shortest path problem in N, we can now use the well-known results for the latter problem to prove needed results about the original problem.

Proposition 1: The set of equations (7) and (6) has exactly one solution. Moreover, these equations can be solved by any dynamic programming algorithm (including Dijkstra's) that solves the shortest path problem in network N.

Proof: It is easy to see that since the graph H is connected, there is a corresponding path from every node in N to the destination v_a.

We naturally assume that the values of all link delays $d_{i,j}$'s are positive. Since clearly $T \geq t_{i,j}$ is also nonnegative, we conclude that all links of N have positive weight. Thus, N contains no nonpositive cycles. It is common knowledge (see, for example, [4]) that under such conditions, there is a unique finite solution to the dynamic programming equations (7) and (6), which determine the lengths of shortest paths to the origin from each of the nodes in the network. Moreover, the fact that all links have positive weights guarantees that the equations can be solved by Dijkstra's algorithm. ■

The above equations can be solved at node a using any centralized shortest-path algorithm, such as Dijkstra's or Bellman-Ford's. The complexity of such algorithms for our problem is the same as that of that of their shortest-path counterparts. For example, Dijkstra's algorithm can solve the problem with time complexity of $O(m+n\log n)$ and space complexity of $O(m)$ [5], where n and m denote the node and edge sets of G, respectively.

Finally, the values of $T_{i,j}$ can now be computed using the formula $T_{i,j} = \max\{T_i, T_j\}$.

Now we are ready to prove that, for each i, the computed value of T_i is the sought out latest time of node a being able to hear from node i. Moreover, our algorithm extracts the maximum amount of information possible, in the sense that it is equivalent to the information obtained through the CF approach. The following theorem makes this precise.

Theorem 1 *The computed values of T_i satisfy the following optimality conditions for every node i:*
(A) Any message of any type that left node i earlier than or at time T_i is guaranteed to have reached us at node a.
(B) All the messages that left node i later than T_i are guaranteed NOT to have reached us.

Proof: The proof is by induction on the order in which the nodes of N are processed by Dijkstra's algorithm. For the first node to be processed, a, the value of $T_a = T$, which, of course, satisfies conditions (A) and (B) of the theorem.

Suppose the first k nodes have been processed. Let node m be an arbitrary unprocessed node node. Then $T_m^{(k)}$, the estimate of the true value of T_m, is defined as follows.

$$T_m^{(k)} = \max\{ \max_{\{j \in PN_H(m): s_{m,j}=1\}}\{T_j - d_{m,j}\}, \max_{\{j \in PN_H(m): s_{m,j}=0\}}\{t_{m,j} - d_{m,j}\}\} \quad (8)$$

Here $PN_H(i)$ denotes the set of nodes in $N_H(i)$ that have already been processed by Dijkstra's algorithm. Let node i be the next node to be processed, namely the node that corresponds to the maximum value of $T_i^{(k)}$ among all unprocessed nodes. By the property of Dijkstra's algorithm, $T_i = T_i^{(k)}$. First, let us prove condition (B). Consider any message M that left node i at time Z_M and reached us by now. We need to show that $Z_M \leq T_i$, which will establish (B) by contraposition. Let P denote one of the paths that M took to reach us (because we are dealing with flooding, there could be more than one such path). Let W denote the subset of nodes of P that were processed by the Dijkstra's algorithm before node i was processed. Consider the order in which M traversed the nodes of P on its way, with node i being the first and node a being the last node. Let node k denote the first node of W in this order. Let l denote the preceding node in this order. We know that M must have arrived at l at some time Z_l, which is greater than or equal to Z_M.

Suppose $s_{l,k} = 1$. Message M had to arrive at node k at time Z_k which is greater than Z_l by at least $d_{l,k}$, i.e., $Z_k \geq Z_l + d_{l,k}$. Because node k has been processed, we know from the inductive hypothesis that if message M left node k after time T_k, this message would not have reached us yet. Thus, $Z_k \leq T_k$. Thus, $Z_l \leq T_k - d_{l,k}$. Because node l has not been processed yet, and node i was "the best choice" (maximized $T_i^{(k)}$) among the unprocessed nodes, we can examine the DP equation (8) to conclude that $T_i \geq T_k - d_{l,k}$. Thus, $Z_l \leq T_i$, which leads immediately to the fact that $Z_M \leq T_i$.

Suppose $s_{l,k} = 0$. If we let Z_k again denote the time that M arrived at k, we know that the link (l,k) must

have been up between the times $Z_k - d_{l,k}$ and Z_k in order for the delivery of that packet to be successful. Let U denote the latest event at or before time Z_k of the link (l,k) going up and t_U denote the time of that event. Thus, link (l,k) was up during the whole period between times t_U and Z_k. Because $t_U \leq Z_k - d_{l,k}$, we conclude that the link has stayed up for at least $d_{l,k}$ seconds and thus node k has generated an update message D reporting event U at time $t_U + d_{l,k}$, which is less than or equal to Z_k. Since this update D was generated no later than the time at which M arrived at k, and since message M was successfully delivered to us along the remaining portion of path P, we conclude that the update D was also successfully delivered to us. Since our records register that $s_{l,k} = 0$, that means that U is not the latest update that we have received, i.e., $t_U + d_{l,k} \leq t_{l,k}$. Since at time Z_k the link was still up, we conclude that $Z_k \leq t_{l,k}$. Since Z_k is at least $d_{l,k}$ units of time later than Z_l, we conclude that $Z_l \leq t_{l,k} - d_{l,k}$. Because node l has not been processed yet, and node i is "the best choice" among the unprocessed nodes, we can examine the DP equation (8) to conclude that $T_i \geq t_{l,k} - d_{l,k}$. Thus, $Z_l \leq T_i$, which leads immediately to the fact that $Z_M \leq T_i$. This finishes the proof of condition (B).

Now the proof of condition (A). Consider any message M that left node i at a time smaller than T_i. From (8), we see that there are two cases.

Case 1. $T_i = T_j - d_{i,j}$, for some $j \in PN_H(i)$ with $s_{i,j} = 1$. We know that the link (i,j) went up at time $t_{i,j}$. Thus, we must have received an update about this event which was generated at time $t_{i,j}+d_{i,j}$, originating from either node i or node j. But we have already proven condition (B), implying that all messages that left node i at time later than T_i are guaranteed NOT to have reached us. Same is true for node j. Thus, no matter which of the two end points originated the update M, we have

$$t_{i,j}+d_{i,j} \leq \max\{T_i, T_j\} = \max\{T_j - d_{i,j}, T_j\} = T_j \quad (9)$$

Since negative changes in (i,j) are reported by j immediately and since (from the inductive hypothesis) we can be sure that node j did not originate any reports of further change in link (i,j) until time T_j, we conclude that the link (i,j) must have been up between the times $T_j - d_{i,j}$ and T_j. Thus, the message M, generated at node i at or before time $T_i = T_j - d_{i,j}$, was successfully delivered to node j at or before time $T_i + d_{i,j} = T_j$, and from the inductive hypothesis we can see that the message went on from node j and was successfully delivered to us.

Case 2. $T_i = t_{i,j} - d_{i,j}$, for some $j \in PN_H(i)$ with $s_{i,j} = 0$. We know that the link (i,j) went down at time $t_{i,j}$. Thus, we must have received an update about this event which was generated at time $t_{i,j}$, originating from either node i or node j. Again, $t_{i,j} \leq \max\{T_i, T_j\} = T_j$, because j was processed before i. Since this update needed to be generated, we conclude that link (i,j) must have been up between the time $t_{i,j} - d_{i,j}$ and time $t_{i,j}$. Thus, the message M, generated at node i at or before time $T_i = t_{i,j} - d_{i,j}$, was successfully delivered to node j at or before time $T_i + d_{i,j} = t_{i,j}$, and from the inductive hypothesis and the fact that $t_{i,j} \leq T_j$, we can see that the message went on from node j and was successfully delivered to us. ■

It is easy to see that Theorem 1 implies

Corollary 1 *Any message that was at node i at or before time T_i is guaranteed to have reached node a by the current time T.*

4 Extrapolating the Values of T_{ij} in the Absence of New Topology Updates

This section presents a method that allows each node to continuously update T_{jk} without having to run the algorithm more than once between any two successive topology updates. This method is based on the fact that T_{jk} changes its slope (as a function of current time T) at most once between two successive topology updates.

The issue we want to consider here is when the above algorithms should be run. We may want to run the algorithm on demand, i.e., every time we need to estimate the state of one of the links. However, in situations when the usage frequency of link state estimates exceeds the frequency of link changes, it would be best to try to run the algorithm on an event-driven basis, i.e., run it only when new link state changes are reported. This approach thus requires us to modify the algorithm in such a way that as the time goes on, we can always get the correct values of $T_{i,j}$ in all cases by looking them up in some simple table and without rerunning the main algorithm, as long as there are no new updates.

One potential difficulty traditionally encountered in this and similar approaches can be illustrated by the following general example. For simplicity, let G be a layered network and let $H = G$. As the current time progresses, the choice of the next "best hop" (from the DP point of view) for each node may change several times. Say, for any node i on layer k, depending on the current time, there will be one of two different next nodes on the "best" path to the destination a. For each of the two next nodes there will be a choice of two nodes after that, and so on, leading to 2^k different paths to

the destination, each the "best" at some point in time. This means that the function describing the value of T_i as a function of current time is likely to be piece-wise linear and consist of 2^k different segments (see Figure 2 for an illustration). This implies exponential look-up table size and exponential computation complexity.

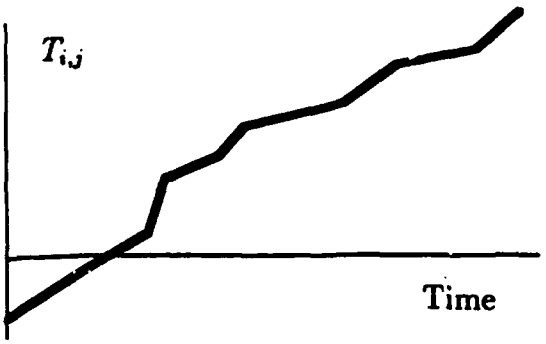


Figure 2: A Possible Shape of the Function Describing the Relationship Between $T_{i,j}$ and T

However, it turns out that in our situation the following proposition is true (see Figure 3 for an illustration):

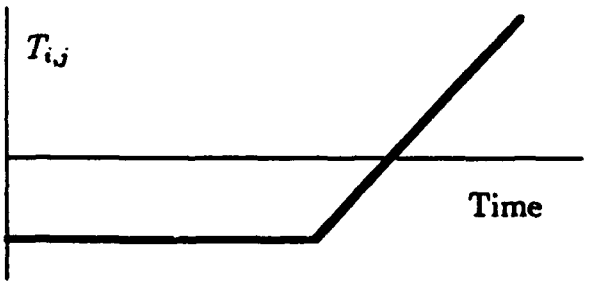


Figure 3: The Actual Shape of the Function Describing the Relationship Between $T_{i,j}$ and T

Proposition 2: Let T_0 denote the latest time that a link-state change occurred in the network. The value of T_i as a function of current time T on the interval $[T_0, \infty)$ is piece-wise linear, consisting of at most 2 different segments.

Proof: An intuitive explanation of the veracity of the proposition is surprisingly simple. The set of paths from a given node i to destination a is divided into two classes. The first class, V, consists of paths with at least one "down" link. If path $P \in V$, then it is easy to see that the latest time that node i could have originated and delivered a message along such a path stays constant regardless of the current time; i.e., this path "freezes in time." Thus, among the paths in class V there is one that is the best in the class for all times. The second class, U, consists of paths with only "up" links. If path $P \in U$, then the latest time that node i could have originated and delivered a message along such a path is equal to $T - D_P$, where D_P denotes the sum of individual link delays on path P. Thus, among the paths in class U, the path with the smallest value of D_P is the best in the class for all times. Thus, the intersection of the linear "time functions" of the two paths that are best in each class will determine the value of T_i as a function of time, proving our proposition.

Here is a more direct proof of the same proposition. Observe that the only links in graph N that depend on the value of T are the links that point to v_a and correspond to a "down" link in G. Let us transform the network N into a new network N' so that all links except one are independent of T. The node set of N' will contain $n+1$ nodes $v_1, v_2, ..., v_a, \bar{v}_a, ...v_n$, with $\bar{v}_a$ being a new node. As before, for every edge $(i,j) \in E(H)$ with $s_{i,j} = 1$, create a directed link (v_i, v_j) of length $d_{i,j}$. For every edge $(i,j) \in E(H)$ with $s_{i,j} = 0$, create a directed link $(v_i, \bar{v}_a)$ of length $T_0 - t_{i,j} + d_{i,j}$. Finally, connect $\bar{v}_a$ to v_a by a link of length $T - T_0$. It is trivial to see that, for any value of $T > T_0$, the graph N' contains only positive edges.

Notice that in N' all the links other than $(\bar{v}_a, v_a)$ have fixed lengths, independent of T. Also notice that $(\bar{v}_a, v_a)$ is the only outgoing link for node $\bar{v}_a$. Let us fix some source node v_i. Let P denote an acyclic path from v_i to v_a, which is shortest for some value of T. Either P does not contain node $\bar{v}_a$, in which case its length is independent of T, or it does, in which case $P = P'(\bar{v}_a, v_a)$, where P' is a shortest path from v_i to $\bar{v}_a$ and is also of fixed length $l(P')$. Thus, $l(P) = l(P') + T - T_0$. Also note that if P is a shortest path from v_i to v_a, then, from the optimality principle, P is a shortest path from v_i to $\bar{v}_a$. Thus, for any value of T, a shortest path $P(T)$ can be chosen as one of the following two paths:

1) A path R_i, which is a shortest path among the set W of all acyclic paths from v_i to v_a that avoid $\bar{v}_a$

2) A path $Q_i = Q_i'(\bar{v}_a, v_a)$, where Q_i' is a shortest path among the set U of all acyclic paths from v_i to $\bar{v}_a$ that avoid v_a.

The set U corresponds to all the paths in G from i to a that contain only "up" links (whose $s_{i,j} = 1$). The set W corresponds to all the paths in G from i to the first encountered "down" link. Since all paths in W and U do not contain link $(\bar{v}_a, v_a)$, they all have fixed lengths, i.e., independent of T. Thus, the paths R_i and Q_i' will be shortest for their corresponding classes, regardless of the value of T. ■

Let us now compare the paths R_i and Q_i. If we let $r_i = l(R_i)$ and $q_i = l(Q_i')$, we see that the path Q_i is shorter than R_i iff $T - T_0 < r_i - q_i$. Thus $D_i = q_i + T - T_0$ for $T \leq T_0 + r_i - q_i$, and $D_i = r_i$ for $T \geq T_0 + r_i - q_i$. Notice that it may be that no path R_i exists, in which case $D_i = q_i + T - T_0$ for all T.

Similarly it may be that no path Q_i exists, in which case $D_i = r_i$ for all T. However, it is easy to check that if node i has a path to a in G, then node v_i will have a path to v_a in N'. Thus, it will always be the case that either R_i or Q_i exists. Recalling that $D_i = T - T_i$, we get $T_i = T_0 - q_i$ for $T \leq T_0 + r_i - q_i$ and $T_i = T - r_i$ for $T \geq T_0 + r_i - q_i$.

We now know how to compute the functions $D_i(T)$, and thus $T_i(T)$. Namely, all we have to do is to compute, for every i simultaneously, the length r_i of the shortest path from v_i to v_a and the length q_i of the shortest path from v_i to $\bar{v}_a$ in the digraph N'', obtained from N' by deleting the "variable-length" link $(\bar{v}_a, v_a)$. Since all the links of N'' are constant, and N'' contains no nonpositive links, this is easily done by running Dijkstra's algorithm twice, first with v_a as the destination and then with $\bar{v}_a$ with the destination.

Thus, we have exhibited an efficient method for computing and caching the values of T_i for the whole future, until new updates arrive. In that case, the update algorithm has to be rerun, not for the whole network, but rather only for those nodes that are affected (i.e., lie behind origin of the update in the shortest-path spanning tree). Efficient methods for updating changes to shortest-path spanning trees are described, for example, in [6].

5 Extensions

In the previous sections, we have described the basic algorithms for computing link state information under event-driven updating under a basic network model. In this section we show how these algorithms can be further improved and can be extended to handle certain changes in the assumptions about the network model.

5.1 Further Reductions in Communication Overhead

When some link, say (i, j), stays down for a while, its endpoints accumulate a lot of update packets waiting to be transmitted over it. By the time the link goes up, however most of these packets actually need not be transmitted and should be discarded. First of all, when there is more than one update from any particular link (k, l), all of these updates, or all but the last one (depending on whether there is an even or odd number of them) should be discarded. But there is a subtler way of discarding even more update packets, which exploits the values of T_j^i and T_i^j that are being computed by the endpoints. Specifically, as soon as link (i, j) goes up, the endpoint i can send the value of T_j^i to node j. Node j now knows that all of the packets that it had received at or before the time T_j^i have reached node j using some alternative path (according to Theorem 1), and thus can be discarded. The only change to our protocols that is required to implement this new idea, is that node j must wait for that extra time $d_{i,j}$ needed for the above exchange, before sending the non-discarded updates, and thus before reporting the link (i, j) as being up. Of course, in most real-life networks the delay of any link at any time is the same in both directions. Thus, it is easy to prove from symmetry that the values of T_j^i and T_i^j are equal to each other and need not even be exchanged.

5.2 Responsibility for Reporting

In this paper, we assumed that when the state of some link (i, j) changes, this change will be reported by both of its endpoints i and j. Thus is the way many networks report such changes. But in many other networks, link state status is reported by only one of the endpoints. The results of this paper easily extend to this case. The only change that need be made is to equation (3). Namely, when the reporting is done by the node $x_{i,j}$, this equation should be replaced by

$$T_{i,j} = T_{x_{i,j}} \text{ for all links } (i, j) . \qquad (10)$$

5.3 Dealing with Variable Delays

In the foregoing discussion, we assumed that the link delay of every "up" link (i, j) is $d_{i,j}$ – constant for all packets at all times. This assumption can be removed and replaced by the assumption that the link delays can vary arbitrarily, but cannot exceed some upper bound, which will now be called $d_{i,j}$. We also need to combine this realistic assumption with a natural policy that (a) if two messages share a portion of a path together, then their order "in the pipe" is not reversed anywhere on that portion; and that (b) if two messages arrive simultaneously for transmission at a node, then the one that has been generated by the node itself is sent first. Under these conditions, it is easy to modify the arguments to show that exactly the same results, formulas, and algorithms as before, will hold. The only exception is condition (B) of Theorem 1, which now should be replaced by the statement that the information obtained with the algorithm is equivalent to the information obtained through the *worst-case delay* execution of the CF CF approach.

It is important to note that without the assumption about the existence of the upper bounds $d_{i,j}$, there will be no way to know for sure about the state of link (i, j) after the time $t_{i,j}$, because, even if all the other

links are "up," there is a chance that an update from (i, j) has been generated but has not arrived yet. Thus, such situations lead us to generalize the deterministic algorithms with statistical estimation and prediction, which is being currently investigated by the authors.

6 Restricting Propagation Distances for Updates

In a very large and highly dynamic network, the communication overhead of distributing updates about every link to all the nodes may be too large. Furthermore, the propagation delay experienced by messages to faraway nodes is likely to be large, providing outdated information about highly dynamic links, which is of little value to the router and can be replaced by steady-state (long-term) statistics. Therefore, it would be useful to limit the propagation of updates to only a relatively small subset of nodes, while ensuring that the error resulting from each node not receiving all the updates is small in some well-defined sense. This section describes a rigorous approach to this subject, and presents a dynamic method of limiting the distance that update messages are propagated.

We will define a limited flooding protocol that automatically allows the radii of propagation to depend on the rates of the individual link dynamics. We will show that, if node a computes the time estimates D_i' according to the algorithm of Section 3 for computing D_i, but using the partial information received with the limited flooding protocol, then $D_i' \leq D_i \leq (1+\epsilon)D_i'$. The limited flooding protocol can therefore be called an ϵ-approximate flooding protocol.

We first need some terminology. We define a *fast* update message to be one that is propagated as quickly as possible, requiring $d_{i,j}$ time to traverse link (i, j). We define a *slow* update message to be one that is (intentionally) propagated more slowly, requiring $(1+\epsilon)d_{i,j}$ to traverse link (i, j). Equivalently, a slow message traverses link (i, j) in $d_{i,j}$ time and then waits at node j for $\epsilon d_{i,j}$ time before it is allowed to leave that node.

We now describe two approximate flooding protocols, F1 and F2. F2 is more efficient than F1 in that it results in a smaller radius of dissemination, but it will be useful to prove the theorem below by first analyzing F1 and then considering how F2 differs from F1. In both flooding protocols F1 and F2, update messages reporting *good* news (a link coming up) are generated and broadcast exactly as in the standard flooding protocol described in Section 3. These messages are therefore fast messages. Update messages reporting *bad* news (a link going down) are generated as in the standard flooding protocol, but are initially propagated as slow messages. In both F1 and F2, a slow message reporting bad news that traverses a link after waiting for it to come up immediately becomes a fast message. In F1, such a message remains a fast message for the rest of its life.

In F2, such a message remains a fast message only until it reaches a node it would have reached, by the same time, had the message never encountered a down link; it then becomes a slow message again. More precisely and more generally, in F2, a message reporting bad news is taken to be a slow message at any given time if and only if the amount of time that has passed since its generation is at most $(1+\epsilon)d(P)$, where P is the path that the message has traversed so far, and $d(P)$ is the sum of the delays of the links of P. Since a slow message requires $(1+\epsilon)d_{i,j}$ to traverse link (i, j), it follows that, in F2, the amount of time that has passed since a bad-news message was generated is never less than $(1+\epsilon)d(P)$. This fact will be used in the proof of the theorem below.

In both F1 and F2, if good news catches up with previous bad news about the same link, then both messages are destroyed. Thus, news about a link that goes down for only a short time reaches only a subset of the nodes. An important property of both protocols, which will be treated more rigorously in the proof of the theorem below, is that a bad-news message that is delayed by having to wait for a link l that went down for a short time, catches up to where it would have been (at the same time) had link l never gone down, before it reaches any node that never receives a message reporting that link l went down. This property allows each node a to correctly estimate D_i, despite the possibility that a message from node i might have been temporarily delayed by a link that went down for a short time without node a ever knowing.

The reason for the asymmetry in F1 and F2 between down links and up links is simple: if an up link goes down for a short amount of time, this only delays messages by that short amount of time, but if a down link comes up for a short amount of time, this may allow messages to reach nodes that could not otherwise be reached at all.

We define $t_{i,j}$ and $s_{i,j}$ for node a as in Section 3, but corresponding to the one of the flooding protocols F1 and F2. We define T_i' and D_i' to be the unique solutions of the dynamic programming equations (4) and (7), respectively, with $t_{i,j}$ and $s_{i,j}$ obtained using F1 or F2. As before, these values can be computed using Dijkstra's algorithm.

Since good news is propagated faster than bad news in the flooding protocols F1 and F2, the topology maintained by node a at any given time is optimistic com-

pared to that maintained by node a using the exact flooding protocol. It is therefore clear that $D'_i \leq D_i$. The following theorem gives the approximate versions of conditions (A) and (B) that are satisfied when F1 or F2 is used, and implies that $D_i \leq (1+\epsilon)D'_i$.

Theorem 2 *Suppose that the values D'_i satisfy equation (7), where $t_{i,j}$ and $s_{i,j}$ are those values maintained by node a at the current time T, using either protocol F1 or F2. Then the following conditions hold:*
(A') Any message that was at node i (and was allowed to leave) at or before time $T-(1+\epsilon)D'_i$ is guaranteed to have reached node a by the current time T, unless the message was destroyed before reaching a.
(B') Any message that was at node i after time $T-D'_i$ is guaranteed NOT to have reached node a by the current time T.

The following corollary ensures that destroyed messages pertain only to links that went down for a short amount of time.

Corollary 2 *Any message generated at node i (about some link (i,j) or (j,i)) at or before time $T-(1+\epsilon)D'_i$ that is destroyed before reaching node a, corresponds to a failure or recovery of the link such that the length of time between the generation of the update reporting the failure and the generation of the update reporting the recovery is at most $\epsilon D'_i$ time units. (Recall that the latter update is generated only after the link has remained up for at least $d_{i,j}$ or $d_{j,i}$ time units.)*

Proof of Theorem 2. Condition (B') follows trivially from condition (B) of Theorem 1 and the inequality $D'_i \leq D_i$. In proving condition (A'), we first assume that the flooding protocol F1 is being used. Thus, if a message containing bad news must wait for a link to come up, then it becomes a fast message for the remainder of its life.

We prove condition (A') by induction on the order in which the nodes of N (see Section 3) are processed by Dijkstra's algorithm. The first node to be processed is a, and we have $D'_a = 0$, which trivially satisfies condition (A'). Let node i be the $k+1$st node processed by Dijkstra's algorithm. Then by equation (7) and Dijkstra's algorithm, there exists a node j processed before node i such that either $s_{i,j} = 1$ and $D'_i = D'_j + d_{i,j}$, or $s_{i,j} = 0$ and $D'_i = T - t_{i,j} + d_{i,j}$.

Suppose a message M (containing either good news or bad news) was ready to leave node i at or before time $T-(1+\epsilon)D'_i$. We will show that M must have reached node a by the current time T.

Case 1. Suppose that $s_{i,j} = 1$ and $D'_i = D'_j + d_{i,j}$. Then we know that link (i,j) went up at time $t_{i,j}$ and stayed up for at least $d_{i,j}$ time units, and that node a has received an update about this event, which was generated at time $t_{i,j} + d_{i,j}$. Since node a received this update by the current time T, condition (B') implies that

$$t_{i,j} + d_{i,j} \leq \max\{T - D'_i, T - D'_j\} = T - D'_j \quad . \quad (11)$$

If $t_{i,j} > T-(1+\epsilon)D'_i$, then message M had to wait for (i,j) to come up and was thus a fast message for the remainder of its life (whether or not M contains bad news). Therefore, M reached and was ready to leave node j by time $t_{i,j} + d_{i,j}$. But the update message corresponding to (i,j) coming up was generated at node j at this time, and arrived at node a by the current time T. Therefore, message M must also have arrived at node a by time T.

Now suppose $t_{i,j} \leq T-(1+\epsilon)D'_i$. If link (i,j) was up between time $T-(1+\epsilon)D'_i$ and time $T-(1+\epsilon)D'_i+d_{i,j}$, then M would reach node j by the latter time, which is equal to $T-(1+\epsilon)D'_j-\epsilon d_{i,j}$, and would thus be ready to leave node j by time $T-(1+\epsilon)D'_j$, whether M is slow or fast. Therefore, by the inductive hypothesis, M must have reached node a by the current time T.

Finally, suppose that $t_{i,j} \leq T-(1+\epsilon)D'_i$, but that link (i,j) was down at some time between time $T-(1+\epsilon)D'_i$ and time $T-(1+\epsilon)D'_i+d_{i,j}$. Then message M had to wait for (i,j) to come up and was thus a fast message for the remainder of its life. Let s be the last time before time $T-(1+\epsilon)D'_i+d_{i,j}$ that (i,j) went down after being up for at least $d_{i,j}$ time units. Then $s \geq t_{i,j} + d_{i,j}$. Let U denote the message generated at node j at time s reporting that link (i,j) went down at time s. Since $s \leq T-(1+\epsilon)D'_i+d_{i,j} \leq T-(1+\epsilon)D'_j$, the inductive hypothesis implies that message U will reach node a by time T if it is not destroyed. Therefore, since message U did not reach node a, it must have been destroyed by a message V generated by node j at some later time $x+d_{i,j}$, reporting that link (i,j) has come back up at time $x > s$. From the definition of s, and the assumption that link (i,j) was down at some time between time $T-(\epsilon)D'_i$ and time $T-(\epsilon)D'_i+d_{i,j}$, it is clear that $x \geq T-(\epsilon)D'_i$. Therefore, since link (i,j) was up between the times x and $x+d_{i,j}$, message M must have arrived at node j by time $x+d_{i,j}$, the same time that message V was generated. Now, since V, and therefore M, was able to catch up with U before U reached node a, and since message U would have reached node a by time T had it not been destroyed, message M must have reached node a by time T.

Case 2. Suppose that $s_{i,j} = 0$ and $D'_i = T-t_{i,j}+d_{i,j}$. Then link (i,j) went down at time $t_{i,j}$, and was the last update node a has received about link (i,j). By

condition (B′) we have

$$t_{i,j} \leq \max\{T - D'_i, T - D'_j\} = T - D'_j \quad , \qquad (12)$$

where the last equality holds because j was processed before i. We know that an update message U, containing the bad news about link (i,j), and generated at time $t_{i,j}$ by either node i or node j, was received by node a by the current time T. We now consider two subcases.

Suppose first that U was generated at node j. Since message U needed to be generated, we conclude that link (i,j) must have been up between time $t_{i,j} - d_{i,j}$ and time $t_{i,j}$. Therefore, message M, generated before time $T-(1+\epsilon)D'_i$, and thus before time $t_{i,j} - d_{i,j} = T - D'_i$, arrived at node j before time $t_{i,j}$. In fact, if M did not have to wait for (i,j) to come up, then it arrived at node j before time $t_{i,j} - \epsilon d_{i,j}$. Therefore, whether M was slow or fast when it traversed link (i,j), it was ready to leave node j at time $t_{i,j}$. Therefore, since U was initially a slow message that was generated at node j at time $t_{i,j}$ and reached node a by time T, message M must have reached node a by time T, whether M was slow or fast when it left node j.

Suppose finally that U was generated at node i. Then since message M was generated at node i before message U, and since U was initially a slow message that reached node a by time T, message M must have reached node a by time T, whether M was slow or fast when it left node i. This proves condition (A′) for protocol F1.

We now prove (A′) for protocol F2. Again suppose a message M (containing either good news or bad news) was ready to leave node i at or before time $T-(1+\epsilon)D'_i$. Let Q be a path in G from i to a defined as follows. Node i is in the path, and for any node i' in the path, link (i',j) is in the path, where j minimizes the right side of the dynamic programming equation (7). It is easily verified that, if we replace the graph G with the subgraph consisting only of links in Q, the D'_i values would not be changed along this path. Therefore, applying condition (A′) for F1 to this subgraph, it follows that, using protocol F1, a copy of M taking path Q would reach a by time T.

Now suppose that j is the first node on path Q at which M would have converted from a fast message to a slow message if protocol F2 were used, and let s be the time that this would happen. (If no such j exists, then protocol F2 behaves exactly as F1, and there is nothing to prove.) Then, letting v denote the node that generated M, and letting $Q_{v,j}$ denote the segment of Q from v to j, by the definition of protocol F2, the amount of time required for M to travel from v to j is at most $(1+\epsilon)d(Q_{v,j})$. But by the discussion following the definition of F2, the amount of time required for M to travel from v to i is at least $(1+\epsilon)d(Q_{v,i})$. If follows that the amount of time required for M to travel from i to j is at most $(1+\epsilon)d(Q_{i,j})$. Therefore, $s - [T - (1+\epsilon)D'_i] \leq (1+\epsilon)d(Q_{i,j})$. It is easily verified that $d(Q_{i,j}) \leq D'_i - D'_j$. Combining these two inequalities, we have $s \leq T - (1+\epsilon)D'_j$. Therefore, repeating the argument starting from j, until no such j exists, it follows that, using F2, M reaches node a by time T. ■

Proof of Corollary 2. Suppose that node i generated a message U at some time $s \leq T - (1+\epsilon)D'_i$ reporting that link l went down, and that the next message V generated by node i about link l (reporting that l had come back up) was generated at some time $x > s + \epsilon D'_i$. It suffices to show that the messages U and V were not destroyed before reaching node a. By condition (A′) of Theorem 2, message U reached node a before time T, unless it was destroyed. By condition (B′) of the theorem, since $x > T - D'_i$, message V will not reach node a until after time T. Therefore message V did not catch up with message U before reaching node a, and so neither message is destroyed. ■

References

[1] D.M. Topkis. Performance analysis of information dissemination by flooding. *IEEE Journal Selected Areas in Comm.*, 7(3):335–340, April 1989.

[2] B Awerbuch and S. Even. Reliable broadcast protocols in unreliable networks. *Networks*, 16:381–396, 1986.

[3] J. Filipiak. *Modelling and control of Dynamic Flows in Communication Networks.* Springer-Verlag, 1988.

[4] E.L. Lawler. *Combinatorial Optimization: Networks and Matroids.* Holt, Rinehart, and Winston, 1976.

[5] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms Proc. 25th Symposium on Foundations of Computer Science, 1984.

[6] I. Richer J.M. McQuillan and E.C. Rosen. The new routing algorithm for the arpanet. *IEEE Trans. Comm.*, COM-28(5):711–719, May 1980.

[7] R.G. Ogier and V. Rutenburg. Minimum-Expected-Delay Alternate Routing. *Proc. IEEE INFOCOM*, 1992.

*U.S. GOVERNMENT PRINTING OFFICE: 1994-510-117-50002

DISTRIBUTION LIST

addresses	number of copies
MR. CHARLES MEYER PL/C38C BLDG #3 525 BROOKS ROAD GRIFFISS AFB NY 13441-4503	5
SRI INTERNATIONAL 333 RAVENSWOOD AVENUE MENLO PARK CA 94025-3493	5
PL/SUL TECHNICAL LIBRARY 26 ELECTRONIC PKY GRIFFISS AFB NY 13441-4514	1
ADMINISTRATOR DEFENSE TECHNICAL INFO CENTER DTIC-FDAC CAMERON STATION BUILDING 5 ALEXANDRIA VA 22304-6145	2
ADVANCED RESEARCH PROJECTS AGENCY 3701 NORTH FAIRFAX DRIVE ARLINGTON VA 22203-1714	1
NAVAL WARFARE ASSESSMENT CENTER GIDEP OPERATIONS CENTER/CODE QA-50 ATTN: E RICHARDS CORONA CA 91718-5000	1
HQ ACC/DRIY ATTN: MAJ. DIVINE LANGLEY AFB VA 23665-5575	1
WRIGHT LABORATORY/AAAI-4 WRIGHT-PATTERSON AFB OH 45433-6543	1

WRIGHT LABORATORY/AAAI-2 1
ATTN: MR FRANKLIN HUTSON
WRIGHT-PATTERSON AFB OH 45433-6543

AFIT/LDEE 1
BUILDING 642, AREA B
WRIGHT-PATTERSON AFB OH 45433-6583

WRIGHT LABORATORY/MTEL 1
WRIGHT-PATTERSON AFB OH 45433

AAMRL/HE 1
WRIGHT-PATTERSON AFB OH 45433-6573

AUL/LSE 1
BLDG 1405
MAXWELL AFB AL 36112-5564

US ARMY STRATEGIC DEF 1
CSSD-IM-PA
PO BOX 1500
HUNTSVILLE AL 35807-3801

COMMANDING OFFICER 1
NAVAL AVIONICS CENTER
LIBRARY D/765
INDIANAPOLIS IN 46219-2189

COMMANDING OFFICER 1
NAVAL OCEAN SYSTEMS CENTER
TECHNICAL LIBRARY
CODE 96428
SAN DIEGO CA 92152-5000

CMDR 1
NAVAL WEAPONS CENTER
TECHNICAL LIBRARY/C3431
CHINA LAKE CA 93555-6001

SPACE & NAVAL WARFARE SYSTEMS COMM
WASHINGTON DC 20363-5100 1

CDR, U.S. ARMY MISSILE COMMAND
REDSTONE SCIENTIFIC INFO CENTER
AMSMI-RD-CS-R/ILL DOCUMENTS
REDSTONE ARSENAL AL 35898-5241 2

ADVISORY GROUP ON ELECTRON DEVICES
ATTN: DOCUMENTS
2011 CRYSTAL DRIVE,SUITE 307
ARLINGTON VA 22202 2

LOS ALAMOS NATIONAL LABORATORY
REPORT LIBRARY
MS 5000
LOS ALAMOS NM 87544 1

AEDC LIBRARY
TECH FILES/MS-100
ARNOLD AFB TN 37389 1

COMMANDER/USAISC
ATTN: ASQP-DO-TL
BLDG 61801
FT HUACHUCA AZ 85613-5000 1

AIR WEATHER SERVICE TECHNICAL LIB
FL 4414
SCOTT AFB IL 62225-5458 1

AFIWC/MSO
102 HALL BLVD STE 315
SAN ANTONIO TX 78243-7016 1

SOFTWARE ENGINEERING INST (SEI)
TECHNICAL LIBRARY
5000 FORBES AVE
PITTSBURGH PA 15213 1

DIRECTOR NSA/CSS 1
W157
9800 SAVAGE ROAD
FORT MEADE MD 21055-6000

NSA 1
E323/HC
SAB2 DOOR 22
FORT MEADE MD 21055-6000

NSA 1
ATTN: D. ALLEY
DIV X911
9800 SAVAGE ROAD
FT MEADE MD 20755-6000

DOD 1
R31
9800 SAVAGE ROAD
FT. MEADE MD 20755-6000

DIRNSA 1
R509
9800 SAVAGE ROAD
FT MEADE MD 20775

DIRECTOR 1
NSA/CSS
R08/R & E BLDG
FORT GEORGE G. MEADE MD 20755-6000

ESC/IC 1
50 GRIFFISS STREET
HANSCOM AFB MA 01731-1619

ESC/AV 1
20 SCHILLING CIRCLE
HANSCOM AFB MA 01731-2816

FL 2807/RESEARCH LIBRARY 1
OL AA/SULL
HANSCOM AFB MA 01731-5000

TECHNICAL REPORTS CENTER 1
MAIL DROP 0130
BURLINGTON ROAD
BEDFORD MA 01731

DEFENSE TECHNOLOGY SEC ADMIN (DTSA) 1
ATTN: STTD/PATRICK SULLIVAN
400 ARMY NAVY DRIVE
SUITE 300
ARLINGTON VA 22202

MISSION

OF

ROME LABORATORY

Mission. The mission of Rome Laboratory is to advance the science and technologies of command, control, communications and intelligence and to transition them into systems to meet customer needs. To achieve this, Rome Lab:

a. Conducts vigorous research, development and test programs in all applicable technologies;

b. Transitions technology to current and future systems to improve operational capability, readiness, and supportability;

c. Provides a full range of technical support to Air Force Materiel Command product centers and other Air Force organizations;

d. Promotes transfer of technology to the private sector;

e. Maintains leading edge technological expertise in the areas of surveillance, communications, command and control, intelligence, reliability science, electro-magnetic technology, photonics, signal processing, and computational science.

The thrust areas of technical competence include: Surveillance, Communications, Command and Control, Intelligence, Signal Processing, Computer Science and Technology, Electromagnetic Technology, Photonics and Reliability Sciences.